MATLAB® 7
Programming Fundamentals
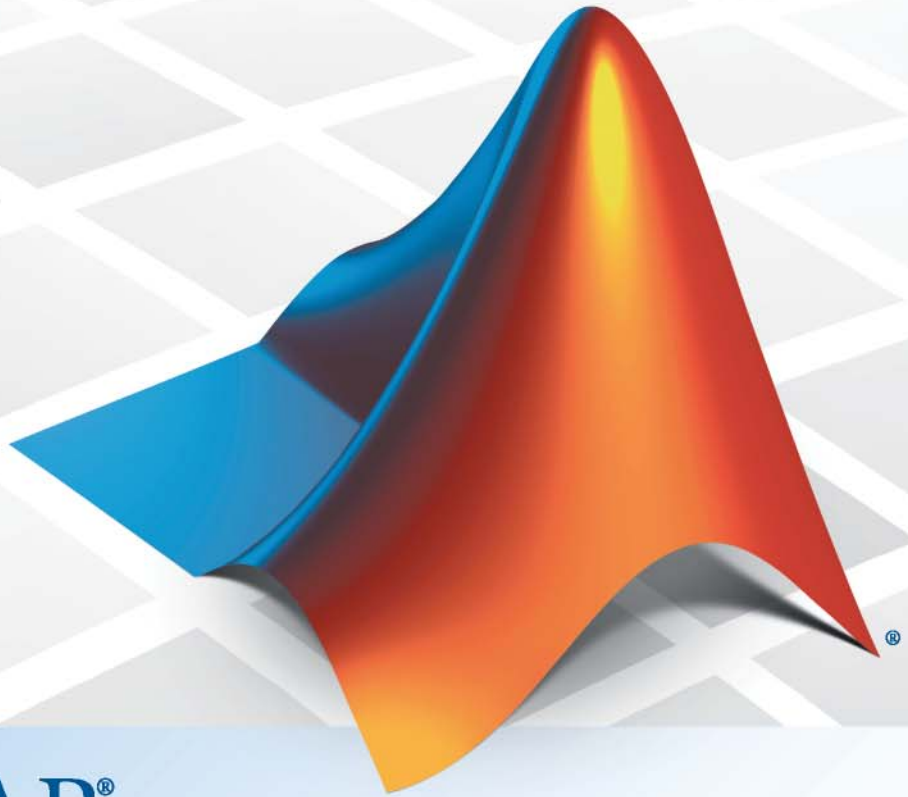
MATLAB®

How to Contact MathWorks

www.mathworks.com   Web
comp.soft-sys.matlab   Newsgroup
www.mathworks.com/contact_TS.html Technical Support

suggest@mathworks.com   Product enhancement suggestions
bugs@mathworks.com   Bug reports
doc@mathworks.com   Documentation error reports
service@mathworks.com   Order status, license renewals, passcodes
info@mathworks.com   Sales, pricing, and general information

508-647-7000 (Phone)

508-647-7001 (Fax)

The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760-2098

For contact information about worldwide offices, see the MathWorks Web site.

MATLAB Programming Fundamentals

© COPYRIGHT 1984–2010 by The MathWorks, Inc.

Trademarks

Patents

Revision History

June 2004	First printing	New for MATLAB 7.0 (Release 14)
October 2004	Online only	Revised for MATLAB 7.0.1 (Release 14SP1)
March 2005	Online only	Revised for MATLAB 7.0.4 (Release 14SP2)
June 2005	Second printing	Minor revision for MATLAB 7.0.4
September 2005	Online only	Revised for MATLAB 7.1 (Release 14SP3)
March 2006	Online only	Revised for MATLAB 7.2 (Release 2006a)
September 2006	Online only	Revised for MATLAB 7.3 (Release 2006b)
March 2007	Online only	Revised for MATLAB 7.4 (Release 2007a)
September 2007	Online only	Revised for Version 7.5 (Release 2007b)
March 2008	Online only	Revised for Version 7.6 (Release 2008a)
October 2008	Online only	Revised for Version 7.7 (Release 2008b)
March 2009	Online only	Revised for Version 7.8 (Release 2009a)
September 2009	Online only	Revised for Version 7.9 (Release 2009b)
March 2010	Online only	Revised for Version 7.10 (Release 2010a)
September 2010	Online only	Revised for Version 7.11 (Release 2010b)

Contents

Classes (Data Types)

1

Basic Program Components

2

Functions and Scripts

3

Types of Functions

4

Using Objects

5

Error Handling

6

Program Scheduling

7

Performance

8

9

Memory Usage

10

Programming Tips

Index

1

Classes (Data Types)

Overview of MATLAB Classes

In this section...
"Fundamental MATLAB Classes" on page 1-2
"How to Use the Different Classes" on page 1-4

Fundamental MATLAB Classes

There are many different data types, or *classes*, that you can work with in the MATLAB® software. You can build matrices and arrays of floating-point and integer data, characters and strings, and logical `true` and `false` states. Function handles connect your code with any MATLAB function regardless of the current scope. Structures and cell arrays, provide a way to store dissimilar types of data in the same array.

There are 15 fundamental classes in MATLAB. Each of these classes is in the form of a matrix or array. This matrix or array is a minimum of 0-by-0 in size and can grow to an n-dimensional array of any size.

All of the fundamental MATLAB classes are circled in the diagram below:
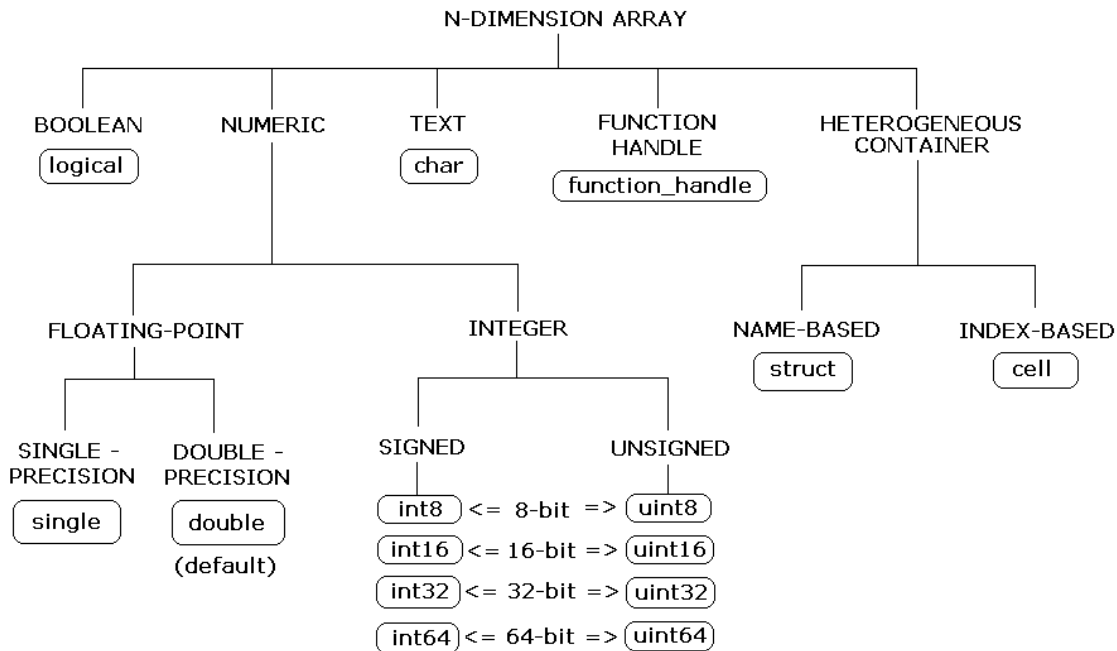
```
                        N-DIMENSION ARRAY

  BOOLEAN      NUMERIC        TEXT        FUNCTION      HETEROGENEOUS
                                           HANDLE         CONTAINER
 ( logical )                ( char )
                                      ( function_handle )

         FLOATING-POINT           INTEGER         NAME-BASED    INDEX-BASED

                                                  ( struct )    ( cell )

   SINGLE -     DOUBLE -      SIGNED        UNSIGNED
   PRECISION    PRECISION

  ( single )   ( double )    ( int8 ) <= 8-bit  => ( uint8 )
                             ( int16 )<= 16-bit => ( uint16 )
               (default)     ( int32 )<= 32-bit => ( uint32 )
                             ( int64 )<= 64-bit => ( uint64 )
```

Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting, reshaping, and mathematical operations.

You can create two-dimensional `double` and `logical` matrices using one of two storage formats: `full` or `sparse`. For matrices with mostly zero-valued elements, a sparse matrix requires a fraction of the storage space required

for an equivalent full matrix. Sparse matrices invoke methods especially tailored to solve sparse problems

These classes require different amounts of storage, the smallest being a `logical` value or 8–bit integer which requires only 1 byte. It is important to keep this minimum size in mind if you work on data in files that were written using a precision smaller than 8 bits.

How to Use the Different Classes

The following table describes these classes in more detail.

Class Name	Documentation	Intended Use
`double`, `single`	Floating-Point Numbers	• Required for fractional numeric data. • Double and Single precision. • Use `realmin` and `realmax` to show range of values. • Two-dimensional arrays can be sparse. • Default numeric type in MATLAB.
`int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`	Integers	• Use for signed and unsigned whole numbers. • More efficient use of memory. • Use `intmin` and `intmax` to show range of values. • Choose from 4 sizes (8, 16, 32, and 64 bits).
`char`	"Characters and Strings" on page 1-35	• Required for text. • Native or unicode. • Converts to/from numeric. • Use with regular expressions. • For multiple strings, use cell arrays.

Class Name	Documentation	Intended Use
logical	Logical Class	• Use in relational conditions or to test state. • Can have one of two values: `true` or `false`. • Also useful in array indexing. • Two-dimensional arrays can be sparse.
function_handle	"Function Handles" on page 1-127	• Pointer to a function. • Enables passing a function to another function • Can also call functions outside usual scope. • Useful in Handle Graphics callbacks. • Save to MAT-file and restore later.
struct	Structures	• Fields store arrays of varying classes and sizes. • Access multiple fields/indices in single operation. • Field names identify contents. • Simple method of passing function arguments. • Use in comma-separated lists for efficiency. • More memory required for overhead
cell	Cell Arrays	• Cells store arrays of varying classes and sizes. • Allows freedom to package data as you want. • Manipulation of elements is similar to arrays. • Simple method of passing function arguments. • Use in comma-separated lists for efficiency. • More memory required for overhead

Numeric Classes

In this section...

Overview

Numeric classes in the MATLAB software include signed and unsigned integers, and single- and double-precision floating-point numbers. By default, MATLAB stores all numeric values as double-precision floating point. (You cannot change the default type and precision.) You can choose to store any number, or array of numbers, as integers or as single-precision. Integer and single-precision arrays offer more memory-efficient storage than double-precision.

All numeric types support basic array operations, such as subscripting, reshaping, and mathematical operations.

Integers

MATLAB has four signed and four unsigned integer classes. Signed types enable you to work with negative integers as well as positive, but cannot represent as wide a range of numbers as the unsigned types because one bit is used to designate a positive or negative sign for the number. Unsigned types give you a wider range of numbers, but these numbers can only be zero or positive.

This section covers:

- "Creating Integer Data" on page 1-7
- "Arithmetic Operations on Integer Classes" on page 1-9
- "Largest and Smallest Values for Integer Classes" on page 1-9
- "Integer Functions" on page 1-10

MATLAB supports 1-, 2-, 4-, and 8-byte storage for integer data. You can save memory and execution time for your programs if you use the smallest integer type that accommodates your data. For example, you do not need a 32-bit integer to store the value 100.

Here are the eight integer classes, the range of values you can store with each type, and the MATLAB conversion function required to create that type:

Class	Range of Values	Conversion Function
Signed 8-bit integer	-2^7 to 2^7-1	int8
Signed 16-bit integer	-2^{15} to $2^{15}-1$	int16
Signed 32-bit integer	-2^{31} to $2^{31}-1$	int32
Signed 64-bit integer	-2^{63} to $2^{63}-1$	int64
Unsigned 8-bit integer	0 to 2^8-1	uint8
Unsigned 16-bit integer	0 to $2^{16}-1$	uint16
Unsigned 32-bit integer	0 to $2^{32}-1$	uint32
Unsigned 64-bit integer	0 to $2^{64}-1$	uint64

Creating Integer Data

MATLAB stores numeric data as double-precision floating point (double) by default. To store data as an integer, you need to convert from double to the desired integer type. Use one of the conversion functions shown in the table above.

For example, to store 325 as a 16-bit signed integer assigned to variable x, type

```
x = int16(325);
```

If the number being converted to an integer has a fractional part, MATLAB rounds to the nearest integer. If the fractional part is exactly `0.5`, then from the two equally nearby integers, MATLAB chooses the one for which the absolute value is larger in magnitude:

```
x = 325.499;                    x = x + .001;

int16(x)                        int16(x)
ans =                           ans =
    325                             326
```

If you need to round a number using a rounding scheme other than the default, MATLAB provides four rounding functions: `round`, `fix`, `floor`, and `ceil`. The `fix` function enables you to override the default and round *towards zero* when there is a nonzero fractional part:

```
x = 325.9;

int16(fix(x))
ans =
    325
```

Arithmetic operations that involve both integers and floating-point always result in an integer data type. MATLAB rounds the result, when necessary, according to the default rounding algorithm. The example below yields an exact answer of `1426.75` which MATLAB then rounds to the next highest integer:

```
int16(325) * 4.39
ans =
    1427
```

The integer conversion functions are also useful when converting other classes, such as strings, to integers:

```
str = 'Hello World';

int8(str)
ans =
    72   101   108   108   111    32    87   111   114   108   100
```

Arithmetic Operations on Integer Classes

MATLAB can perform integer arithmetic on the following types of data:

- Integers or integer arrays of the same integer data type. This yields a result that has the same data type as the operands:

  ```
  x = uint32([132 347 528]) .* uint32(75);
  class(x)
  ans =
     uint32
  ```

- Integers or integer arrays and scalar double-precision floating-point numbers. This yields a result that has the same data type as the integer operands:

  ```
  x = uint32([132 347 528]) .* 75.49;
  class(x)
  ans =
     uint32
  ```

For all binary operations in which one operand is an array of integer data type (except 64-bit integers) and the other is a scalar double, MATLAB computes the operation using elementwise double-precision arithmetic, and then converts the result back to the original integer data type. For binary operations involving a 64-bit integer array and a scalar double, MATLAB computes the operation as if 80-bit extended-precision arithmetic were used, to prevent loss of precision.

For a list of the operations that support integer classes, see Nondouble Data Type Support in the arithmetic operators reference page.

Largest and Smallest Values for Integer Classes

For each integer data type, there is a largest and smallest number that you can represent with that type. The table shown under "Integers" on page 1-6 lists the largest and smallest values for each integer data type in the "Range of Values" column.

You can also obtain these values with the intmax and intmin functions:

```
intmax('int8')                    intmin('int8')
```

```
ans =                                ans =
    127                                  -128
```

If you convert a number that is larger than the maximum value of an integer data type to that type, MATLAB sets it to the maximum value. Similarly, if you convert a number that is smaller than the minimum value of the integer data type, MATLAB sets it to the minimum value. For example,

```
x = int8(300)                        x = int8(-300)
x =                                  x =
    127                                  -128
```

Also, when the result of an arithmetic operation involving integers exceeds the maximum (or minimum) value of the data type, MATLAB sets it to the maximum (or minimum) value:

```
x = int8(100) * 3                    x = int8(-100) * 3
x =                                  x =
    127                                  -128
```

Integer Functions
See Integer Functions on page 1-26 for a list of functions most commonly used with integers in MATLAB.

Floating-Point Numbers
MATLAB represents floating-point numbers in either double-precision or single-precision format. The default is double precision, but you can make any number single precision with a simple conversion function.

This section covers:

- "Double-Precision Floating Point" on page 1-11

- "Single-Precision Floating Point" on page 1-11

- "Creating Floating-Point Data" on page 1-12

- "Arithmetic Operations on Floating-Point Numbers" on page 1-13

- "Largest and Smallest Values for Floating-Point Classes" on page 1-14

- "Accuracy of Floating-Point Data" on page 1-16
- "Avoiding Common Problems with Floating-Point Arithmetic" on page 1-17
- "Floating-Point Functions" on page 1-19
- "References" on page 1-20

Double-Precision Floating Point

MATLAB constructs the double-precision (or `double`) data type according to IEEE® Standard 754 for double precision. Any value stored as a `double` requires 64 bits, formatted as shown in the table below:

Bits	Usage
63	Sign (0 = positive, 1 = negative)
62 to 52	Exponent, biased by 1023
51 to 0	Fraction f of the number 1.f

Single-Precision Floating Point

MATLAB constructs the single-precision (or `single`) data type according to IEEE Standard 754 for single precision. Any value stored as a `single` requires 32 bits, formatted as shown in the table below:

Bits	Usage
31	Sign (0 = positive, 1 = negative)
30 to 23	Exponent, biased by 127
22 to 0	Fraction f of the number 1.f

Because MATLAB stores numbers of type `single` using 32 bits, they require less memory than numbers of type `double`, which use 64 bits. However, because they are stored with fewer bits, numbers of type `single` are represented to less precision than numbers of type `double`.

Creating Floating-Point Data

Use double-precision to store values greater than approximately 3.4×10^{38} or less than approximately -3.4×10^{38}. For numbers that lie between these two limits, you can use either double- or single-precision, but single requires less memory.

Creating Double-Precision Data. Because the default numeric type for MATLAB is double, you can create a double with a simple assignment statement:

```
x = 25.783;
```

The whos function shows that MATLAB has created a 1-by-1 array of type double for the value you just stored in x:

```
whos x
  Name       Size                     Bytes  Class

  x          1x1                          8  double
```

Use isfloat if you just want to verify that x is a floating-point number. This function returns logical 1 (true) if the input is a floating-point number, and logical 0 (false) otherwise:

```
isfloat(x)
ans =
     1
```

You can convert other numeric data, characters or strings, and logical data to double precision using the MATLAB function, double. This example converts a signed integer to double-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = double(y)                      % Convert to double
x =
   -5.8932e+011
```

Creating Single-Precision Data. Because MATLAB stores numeric data as a double by default, you need to use the single conversion function to create a single-precision number:

```
x = single(25.783);
```

The whos function returns the attributes of variable x in a structure. The bytes field of this structure shows that when x is stored as a single, it requires just 4 bytes compared with the 8 bytes to store it as a double:

```
xAttrib = whos('x');
xAttrib.bytes
ans =
     4
```

You can convert other numeric data, characters or strings, and logical data to single precision using the single function. This example converts a signed integer to single-precision floating point:

```
y = int64(-589324077574);          % Create a 64-bit integer

x = single(y)                      % Convert to single
x =
   -5.8932e+011
```

Arithmetic Operations on Floating-Point Numbers

This section describes which classes you can use in arithmetic operations with floating-point numbers.

Double-Precision Operations. You can perform basic arithmetic operations with double and any of the following other classes. When one or more operands is an integer (scalar or array), the double operand must be a scalar. The result is of type double, except where noted otherwise:

- single — The result is of type single

- double

- int* or uint* — The result has the same data type as the integer operand

- char

- logical

This example performs arithmetic on data of types char and double. The result is of type double:

```
c = 'uppercase' - 32;

class(c)
ans =
    double

char(c)
ans =
    UPPERCASE
```

Single-Precision Operations. You can perform basic arithmetic operations with single and any of the following other classes. The result is always single:

- single

- double

- char

- logical

In this example, 7.5 defaults to type double, and the result is of type single:

```
x = single([1.32 3.47 5.28]) .* 7.5;

class(x)
ans =
    single
```

Largest and Smallest Values for Floating-Point Classes

For the double and single classes, there is a largest and smallest number that you can represent with that type.

Largest and Smallest Double-Precision Values. The MATLAB functions realmax and realmin return the maximum and minimum values that you can represent with the double data type:

```
str = 'The range for double is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax, -realmin, realmin, realmax)

ans =
The range for double is:
   -1.79769e+308 to -2.22507e-308 and
    2.22507e-308 to  1.79769e+308
```

Numbers larger than `realmax` or smaller than `-realmax` are assigned the values of positive and negative infinity, respectively:

```
realmax + .0001e+308
ans =
    Inf

-realmax - .0001e+308
ans =
   -Inf
```

Largest and Smallest Single-Precision Values. The MATLAB functions `realmax` and `realmin`, when called with the argument `'single'`, return the maximum and minimum values that you can represent with the `single` data type:

```
str = 'The range for single is:\n\t%g to %g and\n\t %g to  %g';
sprintf(str, -realmax('single'), -realmin('single'), ...
    realmin('single'), realmax('single'))

ans =
The range for single is:
   -3.40282e+038 to -1.17549e-038 and
    1.17549e-038 to  3.40282e+038
```

Numbers larger than `realmax('single')` or smaller than `-realmax ('single')` are assigned the values of positive and negative infinity, respectively:

```
realmax('single') + .0001e+038
ans =
    Inf
```

```
-realmax('single') - .0001e+038
ans =
  -Inf
```

Accuracy of Floating-Point Data

If the result of a floating-point arithmetic computation is not as precise as
you had expected, it is likely caused by the limitations of your computer's
hardware. Probably, your result was a little less exact because the hardware
had insufficient bits to represent the result with perfect accuracy; therefore, it
truncated the resulting value.

Double-Precision Accuracy. Because there are only a finite number
of double-precision numbers, you cannot represent all numbers in
double-precision storage. On any computer, there is a small gap between each
double-precision number and the next larger double-precision number. You
can determine the size of this gap, which limits the precision of your results,
using the eps function. For example, to find the distance between 5 and the
next larger double-precision number, enter

```
format long

eps(5)
ans =
    8.881784197001252e-016
```

This tells you that there are no double-precision numbers between 5 and
5 + eps(5). If a double-precision computation returns the answer 5, the
result is only accurate to within eps(5).

The value of eps(x) depends on x. This example shows that, as x gets larger,
so does eps(x):

```
eps(50)
ans =
    7.105427357601002e-015
```

If you enter eps with no input argument, MATLAB returns the value of
eps(1), the distance from 1 to the next larger double-precision number.

Single-Precision Accuracy. Similarly, there are gaps between any two single-precision numbers. If x has type `single`, `eps(x)` returns the distance between x and the next larger single-precision number. For example,

```
x = single(5);
eps(x)
```

returns

```
ans =
  4.7684e-007
```

Note that this result is larger than `eps(5)`. Because there are fewer single-precision numbers than double-precision numbers, the gaps between the single-precision numbers are larger than the gaps between double-precision numbers. This means that results in single-precision arithmetic are less precise than in double-precision arithmetic.

For a number x of type `double`, `eps(single(x))` gives you an upper bound for the amount that x is rounded when you convert it from `double` to `single`. For example, when you convert the double-precision number `3.14` to `single`, it is rounded by

```
double(single(3.14) - 3.14)
ans =
  1.0490e-007
```

The amount that `3.14` is rounded is less than

```
eps(single(3.14))
ans =
  2.3842e-007
```

Avoiding Common Problems with Floating-Point Arithmetic

Almost all operations in MATLAB are performed in double-precision arithmetic conforming to the IEEE standard 754. Because computers only represent numbers to a finite precision (double precision calls for 52 mantissa bits), computations sometimes yield mathematically nonintuitive results. It is important to note that these results are not bugs in MATLAB.

Use the following examples to help you identify these cases:

Example 1 — Round-Off or What You Get Is Not What You Expect.
The decimal number 4/3 is not exactly representable as a binary fraction. For this reason, the following calculation does not give zero, but rather reveals the quantity eps.

```
e = 1 - 3*(4/3 - 1)

e =
  2.2204e-016
```

Similarly, 0.1 is not exactly representable as a binary number. Thus, you get the following nonintuitive behavior:

```
a = 0.0;
for i = 1:10
  a = a + 0.1;
end
a == 1

ans =
     0
```

Note that the order of operations can matter in the computation:

```
b = 1e-16 + 1 - 1e-16;
c = 1e-16 - 1e-16 + 1;
b == c

ans =
     0
```

There are gaps between floating-point numbers. As the numbers get larger, so do the gaps, as evidenced by:

```
(2^53 + 1) - 2^53

ans =
     0
```

Since pi is not really pi, it is not surprising that sin(pi) is not exactly zero:

```
sin(pi)

ans =
   1.224646799147353e-016
```

Example 2 − Catastrophic Cancellation. When subtractions are performed with nearly equal operands, sometimes cancellation can occur unexpectedly. The following is an example of a cancellation caused by swamping (loss of precision that makes the addition insignificant).

```
sqrt(1e-16 + 1) - 1

ans =
    0
```

Some functions in MATLAB, such as `expm1` and `log1p`, may be used to compensate for the effects of catastrophic cancellation.

Example 3 − Floating-Point Operations and Linear Algebra.
Round-off, cancellation, and other traits of floating-point arithmetic combine to produce startling computations when solving the problems of linear algebra. MATLAB warns that the following matrix A is ill-conditioned, and therefore the system Ax = b may be sensitive to small perturbations:

```
A = diag([2 eps]);
b = [2; eps];
y = A\b;
Warning: Matrix is close to singular or badly scaled.
         Results may be inaccurate. RCOND = 1.110223e-016.
```

These are only a few of the examples showing how IEEE floating-point arithmetic affects computations in MATLAB. Note that all computations performed in IEEE 754 arithmetic are affected, this includes applications written in C or FORTRAN, as well as MATLAB. For more examples and information, see `Technical Note 1108  Common Problems with Floating-Point Arithmetic`.

Floating-Point Functions
See Floating-Point Functions on page 1-26 for a list of functions most commonly used with floating-point numbers in MATLAB.

References

The following references provide more information about floating-point arithmetic.

[1] Moler, Cleve, "Floating Points," *MATLAB News and Notes*, Fall, 1996. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/company/newsletters/news_notes/pdf/Fall96Cleve.pdf

[2] Moler, Cleve, *Numerical Computing with MATLAB*, S.I.A.M. A PDF version is available on the MathWorks Web site at http://www.mathworks.com/moler/.

Complex Numbers

Complex numbers consist of two separate parts: a real part and an imaginary part. The basic imaginary unit is equal to the square root of -1. This is represented in MATLAB by either of two letters: `i` or `j`.

Creating Complex Numbers

The following statement shows one way of creating a complex value in MATLAB. The variable `x` is assigned a complex number with a real part of `2` and an imaginary part of `3`:

```
x = 2 + 3i;
```

Another way to create a complex number is using the `complex` function. This function combines two numeric inputs into a complex output, making the first input real and the second imaginary:

```
x = rand(3) * 5;
y = rand(3) * -8;

z = complex(x, y)
z =
   4.7842 -1.0921i   0.8648 -1.5931i   1.2616 -2.2753i
   2.6130 -0.0941i   4.8987 -2.3898i   4.3787 -3.7538i
   4.4007 -7.1512i   1.3572 -5.2915i   3.6865 -0.5182i
```

You can separate a complex number into its real and imaginary parts using the `real` and `imag` functions:

```
zr = real(z)
zr =
    4.7842    0.8648    1.2616
    2.6130    4.8987    4.3787
    4.4007    1.3572    3.6865

zi = imag(z)
zi =
   -1.0921   -1.5931   -2.2753
   -0.0941   -2.3898   -3.7538
   -7.1512   -5.2915   -0.5182
```

Complex Number Functions

See Complex Number Functions on page 1-27 for a list of functions most commonly used with MATLAB complex numbers in MATLAB.

Infinity and NaN

MATLAB uses the special values `inf`, `-inf`, and `NaN` to represent values that are positive and negative infinity, and not a number respectively.

Infinity

MATLAB represents infinity by the special value `inf`. Infinity results from operations like division by zero and overflow, which lead to results too large to represent as conventional floating-point values. MATLAB also provides a function called `inf` that returns the IEEE arithmetic representation for positive infinity as a `double` scalar value.

Several examples of statements that return positive or negative infinity in MATLAB are shown here.

x = 1/0	x = 1.e1000
x = Inf	x = Inf
x = exp(1000) x = Inf	x = log(0) x = -Inf

Use the isinf function to verify that x is positive or negative infinity:

```
x = log(0);

isinf(x)
ans =
     1
```

NaN

MATLAB represents values that are not real or complex numbers with a special value called NaN, which stands for Not a Number. Expressions like 0/0 and inf/inf result in NaN, as do any arithmetic operations involving a NaN:

```
x = 0/0
x =
   NaN
```

Use the isnan function to verify that the real part of x is NaN:

```
isnan(x)
ans =
     1
```

MATLAB also provides a function called NaN that returns the IEEE arithmetic representation for NaN as a double scalar value:

```
x = NaN;

whos x
  Name      Size                 Bytes  Class
```

```
     x           1x1                         8  double
```

Logical Operations on NaN. Because two NaNs are not equal to each
other, logical operations involving NaN always return false, except for a test
for inequality, (NaN ~= NaN):

```
NaN > NaN
ans =
     0

NaN ~= NaN
ans =
     1
```

Infinity and NaN Functions
See Infinity and NaN Functions on page 1-27 for a list of functions most
commonly used with inf and NaN in MATLAB.

Identifying Numeric Classes
You can check the data type of a variable x using any of these commands.

Command	Operation
whos x	Display the data type of x.
xType = class(x);	Assign the data type of x to a variable.
isnumeric(x)	Determine if x is a numeric type.
isa(x, 'integer') isa(x, 'uint64') isa(x, 'float') isa(x, 'double') isa(x, 'single')	Determine if x is the specified numeric type. (Examples for any integer, unsigned 64-bit integer, any floating point, double precision, and single precision are shown here).
isreal(x)	Determine if x is real or complex.
isnan(x)	Determine if x is Not a Number (NaN).
isinf(x)	Determine if x is infinite.
isfinite(x)	Determine if x is finite.

Display Format for Numeric Values

By default, MATLAB displays numeric output as 5-digit scaled, fixed-point values. You can change the way numeric values are displayed to any of the following:

- 5-digit scaled fixed point, floating point, or the best of the two
- 15-digit scaled fixed point, floating point, or the best of the two
- A ratio of small integers
- Hexadecimal (base 16)
- Bank notation

All available formats are listed on the `format` reference page.

To change the numeric display setting, use either the `format` function or the **Preferences** dialog box (accessible from the MATLAB **File** menu). The `format` function changes the display of numeric values for the duration of a single MATLAB session, while your Preferences settings remain active from one session to the next. These settings affect only how numbers are displayed, not how MATLAB computes or saves them.

Display Format Examples

Here are a few examples of the various formats and the output produced from the following two-element vector x, with components of different magnitudes.

Check the current format setting:

```
get(0, 'format')
ans =
   short
```

Set the value for x and display in 5-digit scaled fixed point:

```
x = [4/3 1.2345e-6]
x =
    1.3333    0.0000
```

Set the format to 5-digit floating point:

```
format short e
x
x =
  1.3333e+000  1.2345e-006
```

Set the format to 15-digit scaled fixed point:

```
format long
x
x =
   1.333333333333333   0.000001234500000
```

Set the format to 'rational' for small integer ratio output:

```
format rational
x
x =
     4/3          1/810045
```

Set an integer value for x and display it in hexadecimal (base 16) format:

```
format hex
x = uint32(876543210)
x =
  343efcea
```

Setting Numeric Format in a Program

To temporarily change the numeric format inside a program, get the original format using the get function and save it in a variable. When you finish working with the new format, you can restore the original format setting using the set function as shown here:

```
origFormat = get(0, 'format');
format('rational');

  -- Work in rational format --

set(0,'format', origFormat);
```

Function Summary

MATLAB provides these functions for working with numeric classes:

- Integer Functions on page 1-26
- Floating-Point Functions on page 1-26
- Complex Number Functions on page 1-27
- Infinity and NaN Functions on page 1-27
- Class Identification Functions on page 1-28
- Output Formatting Functions on page 1-28

Integer Functions

Function	Description
int8, int16, int32, int64	Convert to signed 1-, 2-, 4-, or 8-byte integer.
uint8, uint16, uint32, uint64	Convert to unsigned 1-, 2-, 4-, or 8-byte integer.
ceil	Round towards plus infinity to nearest integer
class	Return the data type of an object.
fix	Round towards zero to nearest integer
floor	Round towards minus infinity to nearest integer
isa	Determine if input value has the specified data type.
isinteger	Determine if input value is an integer array.
isnumeric	Determine if input value is a numeric array.
round	Round towards the nearest integer

Floating-Point Functions

Function	Description
double	Convert to double precision.
single	Convert to single precision.

Floating-Point Functions (Continued)

Function	Description
class	Return the data type of an object.
isa	Determine if input value has the specified data type.
isfloat	Determine if input value is a floating-point array.
isnumeric	Determine if input value is a numeric array.
eps	Return the floating-point relative accuracy. This value is the tolerance MATLAB uses in its calculations.
realmax	Return the largest floating-point number your computer can represent.
realmin	Return the smallest floating-point number your computer can represent.

Complex Number Functions

Function	Description
complex	Construct complex data from real and imaginary components.
i or j	Return the imaginary unit used in constructing complex data.
real	Return the real part of a complex number.
imag	Return the imaginary part of a complex number.
isreal	Determine if a number is real or imaginary.

Infinity and NaN Functions

Function	Description
inf	Return the IEEE value for infinity.
isnan	Detect NaN elements of an array.
isinf	Detect infinite elements of an array.

Infinity and NaN Functions (Continued)

Function	Description
isfinite	Detect finite elements of an array.
nan	Return the IEEE value for Not a Number.

Class Identification Functions

Function	Description
class	Return data type (or class).
isa	Determine if input value is of the specified data type.
isfloat	Determine if input value is a floating-point array.
isinteger	Determine if input value is an integer array.
isnumeric	Determine if input value is a numeric array.
isreal	Determine if input value is real.
whos	Display the data type of input.

Output Formatting Functions

Function	Description
format	Control display format for output.

The Logical Class

Overview of the Logical Class

The logical data type represents a logical true or false state using the numbers 1 and 0, respectively. Certain MATLAB functions and operators return logical true or false to indicate whether a certain condition was found to be true or not. For example, the statement 50>40 returns a logical true value.

Logical data does not have to be scalar; MATLAB supports arrays of logical values as well. For example, the following statement returns a vector of logicals indicating false for the first two elements and true for the last three:

```
[30 40 50 60 70] > 40
ans =
     0     0     1     1     1
```

This statement returns a 4-by-4 array of logical values:

```
x = magic(4) >= 9
x =
     1     0     0     1
     0     1     1     0
     1     0     0     1
     0     1     1     0
```

The MATLAB functions that have names beginning with is (e.g., ischar, issparse) also return a logical value or array:

```
a = [2.5 6.7 9.2 inf 4.8];
```

```
isfinite(a)
ans =
     1     1     1     0     1
```

Logical arrays can also be sparse as long as they have no more than two dimensions:

```
x = sparse(magic(20) > 395)
x =
   (1,1)          1
   (1,4)          1
   (1,5)          1
   (20,18)        1
   (20,19)        1
```

Identifying Logical Arrays

This table shows the commands you can use to determine whether or not an array x is logical. The last function listed, cellfun, operates on cell arrays, which you can read about in the section on cell arrays.

Command	Operation
whos(x)	Display value and data type for x.
islogical(x)	Return true if array is logical.
isa(x, 'logical')	Return true if array is logical.
class(x)	Return string with data type name.
cellfun('islogical', x)	Check each cell array element for logical.

Examples of Identifying Logical Arrays

Create a 3-by-6 array of logicals and use the whos function to identify the size, byte count, and class (i.e., data type) of the array.

```
% Initialize the state of the random number generator.
rand('state',0);
A = rand(3,6) > .5
A =
```

```
   1     0     0     0     1     0
   0     1     0     1     1     1
   1     1     1     1     0     1

whos A
  Name      Size              Bytes  Class      Attributes

   A        3x6                  18  logical
```

Find the class of each of these expressions:

```
B = logical(-2.8);  C = false;  D = 50>40;  E = isinteger(4.9);

whos B C D E
  Name      Size              Bytes  Class      Attributes

   B        1x1                   1  logical
   C        1x1                   1  logical
   D        1x1                   1  logical
   E        1x1                   1  logical
```

Display the class of A:

```
% Initialize the state of the random number generator.
rand('state',0);
A = rand(3,6) > .5

fprintf('A is a %s\n', class(A))
   A is a logical
```

Create cell array C and use islogical to identify the logical elements:

```
C = {1, 0, true, false, pi, A};
cellfun('islogical', C)
ans =
     0     0     1     1     0     1
```

Functions that Return a Logical Result

This table shows some of the MATLAB operations that return a logical true or false. Most mathematics operations are not supported on logical values.

Function	Operation
true, false	Setting value to true or false
logical	Numeric to logical conversion
& (and), | (or), ~ (not), xor, any, all	Logical operations
&&, ||	Short-circuit AND and OR
== (eq), ~= (ne), < (lt), > (gt), <= (le), >= (ge)	Relational operations
All is* functions, cellfun	Test operations
strcmp, strncmp, strcmpi, strncmpi	String comparisons

Examples of Functions that Return a Logical Result

MATLAB functions that test the state of a variable or expression return a logical result:

```
A = isstrprop('abc123def', 'alpha')
A =
     1    1    1    0    0    0    1    1    1
```

Logical functions such as xor return a logical result:

```
xor([1 0 'ab' 2.4], [ 0 0 'ab', 0])
ans =
     1    0    0    1
```

Note however that the bitwise operators do not return a logical:

```
X = bitxor(3, 12);
whos X
  Name      Size            Bytes  Class     Attributes

  X         1x1                 8  double
```

String comparison functions also return a logical:

```
S = 'D:\matlab\mfiles\test19.m';
strncmp(S, 'D:\matlab', 9)
ans =
```

```
        1
```

Note the difference between the elementwise and short-circuit logical operators. Short-circuit operators, such as `&&` and `||`, test only as much of the input expression as necessary. In the second part of this example, it makes no difference that B is undefined because the state of A alone determines that the expression is `false`:

```
A = 0;
A & B
??? Undefined function or variable 'B'.

A && B
ans =
     0
```

One way of implementing an infinite loop is to use the `while` function along with the logical constant `true`:

```
while true
a = [];  b = [];
a = input('Enter username: ', 's');

   if ~isempty(a)
   b = input('Enter password: ', 's');
   end

   if ~isempty(b)
   disp 'Attempting to log in to account ...'
   break
   end
end
```

Using Logical Arrays in Conditional Statements

Conditional statements are useful when you want to execute a block of code only when a certain condition is met. For example, the `sprintf` command shown below is valid only if `str` is a nonempty string:

```
str = input('Enter input string: ', 's');
if ~isempty(str) && ischar(str)
   sprintf('Input string is ''%s''', str)
```

```
    end
```

Now run the code:

```
Enter input string: Hello
ans =
    Input string is 'Hello'
```

Using Logical Arrays in Indexing

A logical matrix provides a different type of array indexing in MATLAB. While most indices are numeric, indicating a certain row or column number, logical indices are positional. That is, it is the *position* of each 1 in the logical matrix that determines which array element is being referred to.

See "Using Logicals in Array Indexing" for more information on this subject.

Characters and Strings

Creating Character Arrays

A character in the MATLAB software is actually an integer value converted to its Unicode® UTF-16 character equivalent. A character string is a vector with components that are the numeric codes for the characters.

The elements of a character or string belong to the char class. Arrays of class char can hold multiple strings, as long as each string in the array has the same length. (This is because MATLAB arrays must be rectangular.) To store an array of strings of unequal length, use a cell array.

Creating a Single Character

Store a single character in the MATLAB workspace by enclosing the character in single quotation marks and assigning it to a variable:

```
hChar = 'h';
```

This creates a 1-by-1 matrix of class char. Each character occupies 2 bytes of workspace memory:

```
whos hChar
  Name       Size            Bytes  Class     Attributes
```

```
    hChar       1x1                     2  char
```

The numeric value of hChar is 104:

```
uint8(hChar)
ans =
   104
```

Creating a Character String

Create a string by enclosing a sequence of letters in single quotation marks. MATLAB represents the five-character string shown below as a 1-by-5 vector of class char. It occupies 2 bytes of memory for each character in the string:

```
str = 'Hello';

whos str
  Name        Size              Bytes  Class     Attributes

  str         1x5                  10  char
```

The uint8 function converts characters to their numeric values:

```
str_numeric = uint8(str)
str_numeric =
    72  101  108  108  111
```

The char function converts the integer vector back to characters:

```
str_alpha = char([72 101 108 108 111])
str_alpha =
    Hello
```

Creating a Rectangular Character Array

You can join two or more strings together to create a new character array. This is called *concatenation* and is explained for numeric arrays in the section "Concatenating Matrices". As with numeric arrays, you can combine character arrays vertically or horizontally to create a new character array.

Alternatively, combine strings into a cell array. Cell arrays are flexible containers that allow you to easily combine strings of varying length.

Combining Strings Vertically. To combine strings into a two-dimensional character array, use either of these methods:

- Apply the MATLAB concatenation operator, []. Separate each row with a semicolon (;). Each row must contain the same number of characters. For example, combine three strings of equal length:

```
dev_title = ['Thomas R. Lee'; ...
             'Sr. Developer'; ...
             'SFTware Corp.'];
```

If the strings have different lengths, pad with space characters as needed. For example:

```
mgr_title = ['Harold A. Jorgensen      '; ...
             'Assistant Project Manager'; ...
             'SFTware Corp.            '];
```

- Call the char function. If the strings are different length, char pads the shorter strings with trailing blanks so that each row has the same number of characters. For example, combine three strings of different lengths:

```
mgr_title = char('Harold A. Jorgensen', ...
    'Assistant Project Manager', 'SFTware Corp.');
```

The char function creates a 3-by-25 character array mgr_title.

Combining Strings Horizontally. To combine strings into a single row vector, use either of these methods:

- Apply the MATLAB concatenation operator, []. Separate the input strings with a comma or a space. This method preserves any trailing spaces in the input arrays. For example, combine several strings:

```
name =    'Thomas R. Lee';
title =   'Sr. Developer';
company = 'SFTware Corp.';
```

```
    full_name = [name ', ' title ', ' company]
```

MATLAB returns

```
full_name =
   Thomas R. Lee, Sr. Developer, SFTware Corp.
```

- Call the string concatenation function, strcat. This method removes trailing spaces in the inputs. For example, combine strings to create a hypothetical e-mail address:

```
name   = 'myname   ';
domain = 'mydomain ';
ext    = 'com      ';

address = strcat(name, '@', domain, '.', ext)
```

MATLAB returns

```
address =
   myname@mydomain.com
```

Identifying Characters in a String

Use any of the following functions to identify a character or string, or certain characters in a string:

Function	Description
ischar	Determine whether the input is a character array.
isletter	Find all alphabetic letters in the input string.
isspace	Find all space characters in the input string.
isstrprop	Find all characters of a specific category.

```
str = 'Find the space characters in this string';
%         |   |     |          | |    |
%         5   9     15         26 29   34

find(isspace(str))
ans =
```

```
    5     9    15    26    29    34
```

Working with Space Characters

The `blanks` function creates a string of space characters. The following example creates a string of 15 space characters:

```
s = blanks(15)
s =
```

To make the example more useful, append a `'|'` character to the beginning and end of the blank string so that you can see the output:

```
['|' s '|']          % Make result visible.
ans =
   |               |
```

Insert a few nonspace characters in the middle of the blank string:

```
s(6:10) = 'AAAAA';

['|' s '|']          % Make result visible.
ans =
   |     AAAAA     |
```

You can justify the positioning of these characters to the left or right using the `strjust` function:

```
sLeft = strjust(s, 'left');

['|' sLeft '|']         % Make result visible.
ans =
   |AAAAA          |

sRight = strjust(s, 'right');

['|' sRight '|']         % Make result visible.
ans =
   |          AAAAA|
```

Remove all trailing space characters with `deblank`:

```
sDeblank = deblank(s);

['|' sDeblank '|']          % Make result visible.
ans =
    |       AAAAA|
```

Remove all leading and trailing spaces with `strtrim`:

```
sTrim = strtrim(s);

['|' sTrim '|']             % Make result visible.
ans =
    |AAAAA|
```

Expanding Character Arrays

Generally, MathWorks® does not recommend expanding the size of an existing character array by assigning additional characters to indices beyond the bounds of the array such that part of the array becomes padded with zeros.

Cell Arrays of Strings

Creating strings in a regular MATLAB array requires that all strings in the array be of the same length. This often means that you have to pad blanks at the end of strings to equalize their length. However, another type of MATLAB array, the cell array, can hold different sizes and types of data in an array without padding. Cell arrays provide a more flexible way to store strings of varying length.

For details on cell arrays, see Cell Arrays in the Programming Fundamentals documentation.

Converting to a Cell Array of Strings

The `cellstr` function converts a character array into a cell array of strings. Consider this character array:

```
data = ['Allison Jones';'Development  ';'Phoenix      '];
```

Each row of the matrix is padded so that all have equal length (in this case, 13 characters).

Now use `cellstr` to create a column vector of cells, each cell containing one of the strings from the `data` array:

```
celldata = cellstr(data)
celldata =
    'Allison Jones'
    'Development'
    'Phoenix'
```

Note that the `cellstr` function strips off the blanks that pad the rows of the input string matrix:

```
length(celldata{3})
ans =
    7
```

The `iscellstr` function determines if the input argument is a cell array of strings. It returns a logical 1 (`true`) in the case of `celldata`:

```
iscellstr(celldata)
ans =
    1
```

Use `char` to convert back to a standard padded character array:

```
strings = char(celldata)
strings =
   Allison Jones
   Development
   Phoenix

length(strings(3,:))
ans =
    13
```

Functions for Cell Arrays of Strings

This table describes the MATLAB functions for working with cell arrays.

Function	Description
cellstr	Convert a character array to a cell array of strings.
char	Convert a cell array of strings to a character array.
deblank	Remove trailing blanks from a string.
iscellstr	Return true for a cell array of strings.
sort	Sort elements in ascending or descending order.
strcat	Concatenate strings.
strcmp	Compare strings.

You can also use the following set functions with cell arrays of strings.

Function	Description
intersect	Set the intersection of two vectors.
ismember	Detect members of a set.
setdiff	Return the set difference of two vectors.
setxor	Set the exclusive OR of two vectors.
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

Formatting Strings

The following MATLAB functions offer the capability to compose a string that includes ordinary text and data formatted to your specification:

- sprintf — Write formatted data to an output string

- fprintf — Write formatted data to an output file or the Command Window

- warning — Display formatted data in a warning message

- error — Display formatted data in an error message and abort

- assert — Generate an error when a condition is violated

- MException — Capture error information

The syntax of each of these functions includes formatting operators similar to those used by the `printf` function in the C programming language. For example, `%s` tells MATLAB to interpret an input value as a string, and `%d` means to format an integer using decimal notation.

The general formatting syntax for these functions is

```
functionname(..., format_string, value1, value2, ..., valueN)
```

where the `format_string` argument expresses the basic content and formatting of the final output string, and the `value` arguments that follow supply data values to be inserted into the string.

Here is a sample `sprintf` statement, also showing the resulting output string:

```
sprintf('The price of %s on %d/%d/%d was $%.2f.', ...
        'bread', 7, 1, 2006, 2.49)
ans =
    The price of bread on 7/1/2006 was $2.49.
```

The following sections cover

- "The Format String" on page 1-44
- "Input Value Arguments" on page 1-45
- "The Formatting Operator" on page 1-46
- "Constructing the Formatting Operator" on page 1-47
- "Setting Field Width and Precision" on page 1-52
- "Restrictions for Using Identifiers" on page 1-55

Note The examples in this section of the documentation use only the `sprintf` function to demonstrate how string formatting works. However, you can run the examples using the `fprintf`, `warning`, and `error` functions as well.

The Format String

The first input argument in the sprintf statement shown above is the format_string:

```
'The price of %s on %d/%d/%d was $%.2f.'
```

This argument can include ordinary text, formatting operators and, in some cases, special characters. The formatting operators for this particular string are: %s, %d, %d, %d, and %.2f.

Following the format_string argument are five additional input arguments, one for each of the formatting operators in the string:

```
'bread', 7, 1, 2006, 2.49
```

When MATLAB processes the format string, it replaces each operator with one of these input values.

Special Characters. Special characters are a part of the text in the string. But, because they cannot be entered as ordinary text, they require a unique character sequence to represent them. Use any of the following character sequences to insert special characters into the output string.

To Insert a . . .	Use . . .
Single quotation mark	''
Percent character	%%
Backslash	\\
Alarm	\a
Backspace	\b
Form feed	\f
New line	\n
Carriage return	\r
Horizontal tab	\t
Vertical tab	\v

To Insert a . . .	Use . . .
Hexadecimal number, N	\xN
Octal number, N	\N

Input Value Arguments

In the syntax

```
functionname(..., format_string, value1, value2, ..., valueN)
```

The value arguments must immediately follow format_string in the argument list. In most instances, you supply one of these value arguments for each formatting operator used in the format_string. Scalars, vectors, and numeric and character arrays are valid value arguments. You cannot use cell arrays or structures.

If you include fewer formatting operators than there are values to insert, MATLAB reuses the operators on the additional values. This example shows two formatting operators and six values to insert into the string:

```
sprintf('%s = %d\n', 'A', 479, 'B', 352, 'C', 651)
ans =
    A = 479
    B = 352
    C = 651
```

You can also specify multiple value arguments as a vector or matrix. The format_string needs one %s operator for the entire matrix or vector:

```
mvec = [77 65 84 76 65 66];

sprintf('%s ', char(mvec))
ans =
    MATLAB
```

Sequential and Numbered Argument Specification.

You can place value arguments in the argument list either sequentially (that is, in the same order in which their formatting operators appear in the string),

or by identifier (adding a number to each operator that identifies which value argument to replace it with). By default, MATLAB uses sequential ordering.

To specify arguments by a numeric identifier, add a positive integer followed by a $ sign immediately after the % sign in the operator. Numbered argument specification is explained in more detail under the topic "Value Identifiers" on page 1-52.

Ordered Sequentially	Ordered By Identifier
```sprintf('%s %s %s', ... '1st', '2nd', '3rd') ans = 1st 2nd 3rd```	```sprintf('%3$s %2$s %1$s', ... '1st', '2nd', '3rd') ans = 3rd 2nd 1st```

The Formatting Operator

Formatting operators tell MATLAB how to format the numeric or character value arguments and where to insert them into the string. These operators control the notation, alignment, significant digits, field width, and other aspects of the output string.

A formatting operator begins with a % character, which may be followed by a series of one or more numbers, characters, or symbols, each playing a role in further defining the format of the insertion value. The final entry in this series is a single *conversion character* that MATLAB uses to define the notation style for the inserted data. Conversion characters used in MATLAB are based on those used by the printf function in the C programming language.

Here is a simple example showing five formatting variations for a common value:

```
A = pi*100*ones(1,5);

sprintf(' %f \n %.2f \n %+.2f \n %12.2f \n %012.2f \n', A)
ans =
    314.159265      % Display in fixed-point notation (%f)
    314.16          % Display 2 decimal digits (%.2f)
    +314.16         % Display + for positive numbers (%+.2f)
          314.16    % Set width to 12 characters (%12.2f)
```

```
000000314.16     % Replace leading spaces with 0 (%012.2f)
```

Constructing the Formatting Operator

The fields that make up a formatting operator in MATLAB are as shown here, in the order they appear from right to left in the operator. The rightmost field, the conversion character, is required; the five to the left of that are optional. Each of these fields is explained in a section below:

- Conversion Character — Specifies the notation of the output.

- Subtype — Further specifies any nonstandard types.

- Precision — Sets the number of digits to display to the right of the decimal point, or the number of significant digits to display.

- Field Width — Sets the minimum number of digits to display.

- Flags — Controls the alignment, padding, and inclusion of plus or minus signs.

- Value Identifiers — Map formatting operators to value input arguments. Use the identifier field when value arguments are not in a sequential order in the argument list.

Here is an example of a formatting operator that uses all six fields. (Space characters are not allowed in the operator. They are shown here only to improve readability of the figure).
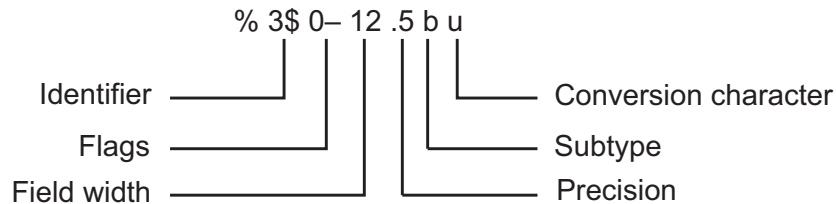
An alternate syntax, that enables you to supply values for the field width and precision fields from values in the argument list, is shown below. See the section "Specifying Field Width and Precision Outside the format String" on page 1-54 for information on when and how to use this syntax. (Again, space characters are shown only to improve readability of the figure.)
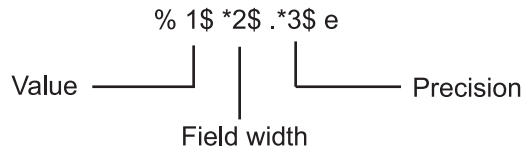
Each field of the formatting operator is described in the following sections. These fields are covered as they appear going from right to left in the formatting operator, starting with the Conversion Character and ending with the Identifier field.

Conversion Character. The conversion character specifies the notation of the output. It consists of a single character and appears last in the format specifier. It is the only required field of the format specifier other than the leading % character.

Specifier	Description
c	Single character
d	Decimal notation (signed)
e	Exponential notation (using a lowercase e as in 3.1415e+00)
E	Exponential notation (using an uppercase E as in 3.1415E+00)
f	Fixed-point notation
g	The more compact of %e or %f. (Insignificant zeros do not print.)
G	Same as %g, but using an uppercase E
o	Octal notation (unsigned)
s	String of characters
u	Decimal notation (unsigned)
x	Hexadecimal notation (using lowercase letters a–f)
X	Hexadecimal notation (using uppercase letters A–F)

This example uses conversion characters to display the number 46 in decimal, fixed-point, exponential, and hexadecimal formats:

```
A = 46*ones(1,4);

sprintf('%d   %f   %e   %X', A)
ans =
    46   46.000000   4.600000e+001   2E
```

Subtype. The subtype field is a single alphabetic character that immediately precedes the conversion character. The following nonstandard subtype specifiers are supported for the conversion characters %o, %x, %X, and %u.

b	The underlying C data type is a double rather than an unsigned integer. For example, to print a double-precision value in hexadecimal, use a format like '%bx'.
t	The underlying C data type is a float rather than an unsigned integer.

To specify the number of bits for the conversion of an integer value (corresponding to conversion characters %d, %i, %u, %o, %x, or %X), use one of the following subtypes.

l	64-bit value.
h	16-bit value.

Precision. precision in a formatting operator is a nonnegative integer that immediately follows a period. For example, the specifier %7.3f, has a precision of 3. For the %g specifier, precision indicates the number of significant digits to display. For the %f, %e, and %E specifiers, precision indicates how many digits to display to the right of the decimal point.

Here are some examples of how the precision field affects different types of notation:

```
sprintf('%g   %.2g   %f   %.2f', pi*50*ones(1,4))
ans =
    157.08   1.6e+002   157.079633   157.08
```

Precision is not usually used in format specifiers for strings (i.e., %s). If you do use it on a string and if the value p in the precision field is less than the

number of characters in the string, MATLAB displays only p characters of the string and truncates the rest.

You can also supply the value for a precision field from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 1-54 for more information on this.

For more information on the use of precision in formatting, see "Setting Field Width and Precision" on page 1-52.

Field Width. Field width in a formatting operator is a nonnegative integer that tells MATLAB the minimum number of digits or characters to use when formatting the corresponding input value. For example, the specifier %7.3f, has a width of 7.

Here are some examples of how the width field affects different types of notation:

```
sprintf('|%e|%15e|%f|%15f|', pi*50*ones(1,4))
ans =
    |1.570796e+002|  1.570796e+002|157.079633|      157.079633|
```

When used on a string, the field width can determine whether MATLAB pads the string with spaces. If width is less than or equal to the number of characters in the string, it has no effect.

```
sprintf('%30s', 'Pad left with spaces')
ans =
          Pad left with spaces
```

You can also supply a value for field width from outside of the format specifier. See the section "Specifying Field Width and Precision Outside the format String" on page 1-54 for more information on this.

For more information on the use of field width in formatting, see "Setting Field Width and Precision" on page 1-52.

Flags. You can control the output using any of these optional flags:

Character	Description	Example
A minus sign (-)	Left-justifies the converted argument in its field.	%-5.2d
A plus sign (+)	Always prints a sign character (+ or –).	%+5.2d
A space ( )	Inserts a space before the value.	% 5.2f
Zero (0)	Pads with zeros rather than spaces.	%05.2f
A pound sign (#)	Modifies selected numeric conversions: • For %o, %x, or %X, print 0, 0x, or 0X prefix. • For %f, %e, or %E, print decimal point even when precision is 0. • For %g or %G, do not remove trailing zeros or decimal point.	%#5.0f

Right- and left-justify the output. The default is to right-justify:

```
sprintf('right-justify: %12.2f\nleft-justify: %-12.2f', ...
        12.3, 12.3)
ans =
    right-justify:        12.30
    left-justify: 12.30
```

Display a + sign for positive numbers. The default is to omit the + sign:

```
sprintf('no sign: %12.2f\nsign: %+12.2f', ...
        12.3, 12.3)
ans =
```

```
no sign:         12.30
sign:        +12.30
```

Pad to the left with spaces or zeros. The default is to use space-padding:

```
sprintf('space-padded: %12.2f\nzero-padded: %012.2f', ...
        5.2, 5.2)
ans =
    space-padded:         5.20
    zero-padded: 000000005.20
```

Note You can specify more than one flag in a formatting operator.

Value Identifiers. By default, MATLAB inserts data values from the argument list into the string in a sequential order. If you have a need to use the value arguments in a nonsequential order, you can override the default by using a numeric identifier in each format specifier. Specify nonsequential arguments with an integer immediately following the % sign, followed by a $ sign.

Ordered Sequentially	Ordered By Identifier
```sprintf('%s %s %s', ... '1st', '2nd', '3rd') ans = 1st 2nd 3rd```	```sprintf('%3$s %2$s %1$s', ... '1st', '2nd', '3rd') ans = 3rd 2nd 1st```

Setting Field Width and Precision

This section provides further information on the use of the field width and precision fields of the formatting operator:

- "Effect on the Output String" on page 1-53

- "Specifying Field Width and Precision Outside the format String" on page 1-54

- "Using Identifiers In the Width and Precision Fields" on page 1-54

Effect on the Output String. The figure below illustrates the way in which the field width and precision settings affect the output of the string formatting functions. In this figure, the zero following the % sign in the formatting operator means to add leading zeros to the output string rather than space characters:

Whole part of input
value has has 3 digits

Result has w digits,
extending to the
left with zeros

Format operator

123.45678 ⟶ %09.3f ⟶ 00123.457

field width: w = 9
precision: p = 3

Fractional part of input
value has 5 digits

Fractional part of the
result has p digits
and is rounded
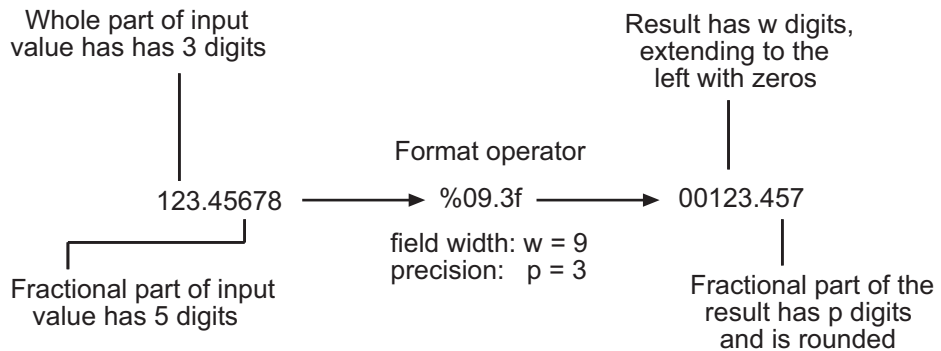
General rules for formatting

- If precision is not specified, it defaults to 6.

- If precision (p) is less than the number of digits in the fractional part of the input value (f), then only p digits are shown to the right of the decimal point in the output, and that fractional value is rounded.

- If precision (p) is greater than the number of digits in the fractional part of the input value (f), then p digits are shown to the right of the decimal point in the output, and the fractional part is extended to the right with p-f zeros.

- If field width is not specified, it defaults to precision + 1 + the number of digits in the whole part of the input value.

- If field width (w) is greater than p+1 plus the number of digits in the whole part of the input value (n), then the whole part of the output value is extended to the left with w-(n+1+p) space characters or zeros, depending on whether or not the zero flag is set in the Flags field. The default is to extend the whole part of the output with space characters.

Specifying Field Width and Precision Outside the format String. To specify field width or precision using values from a sequential argument list, use an asterisk (`*`) in place of the `field width` or `precision` field of the formatting operator.

This example formats and displays three numbers. The formatting operator for the first, `%*f`, has an asterisk in the field width location of the formatting operator, specifying that just the field width, 15, is to be taken from the argument list. The second operator, `%.*f` puts the asterisk after the decimal point meaning, that it is the precision that is to take its value from the argument list. And the third operator, `%*.*f`, specifies both field width and precision in the argument list:

```
  sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...    % Width for 123.45678 is 15
        3, 16.42837, ...      % Precision for rand*20 is .3
        6, 4, pi)             % Width & Precision for pi is 6.4
  ans =
      123.456780   16.428   3.1416
```

You can mix the two styles. For example, this statement gets the field width from the argument list and the precision from the format string:

```
  sprintf('%*.2f', 5, 123.45678)
  ans =
      123.46
```

Using Identifiers In the Width and Precision Fields. You can also derive field width and precision values from a nonsequential (i.e., numbered) argument list. Inside the formatting operator, specify `field width` and/or `precision` with an asterisk followed by an identifier number, followed by a $ sign.

This example from the previous section shows how to obtain field width and precision from a sequential argument list:

```
  sprintf('%*f   %.*f   %*.*f', ...
        15, 123.45678, ...
        3, 16.42837, ...
        6, 4, pi)
```

```
ans =
     123.456780   16.428   3.1416
```

Here is an example of how to do the same thing using numbered ordering. Field width for the first output value is 15, precision for the second value is 3, and field width and precision for the third value is 6 and 4, respectively. If you specify field width or precision with identifiers, then you must specify the value with an identifier as well:

```
sprintf('%1$*4$f   %2$.*5$f   %3$*6$.*7$f', ...
123.45678, 16.42837, pi, 15, 3, 6, 4)

ans =
     123.456780   16.428   3.1416
```

Restrictions for Using Identifiers

If any of the formatting operators in a string include an identifier field, then all of the operators in that string must do the same; you cannot use both sequential and nonsequential ordering in the same function call.

Valid Syntax	Invalid Syntax
```sprintf('%d %d %d %d', ... 1, 2, 3, 4) ans = 1 2 3 4```	```sprintf('%d %3$d %d %d', ... 1, 2, 3, 4) ans = 1```

If your command provides more value arguments than there are formatting operators in the format string, MATLAB reuses the operators. However, MATLAB allows this only for commands that use sequential ordering. You cannot reuse formatting operators when making a function call with numbered ordering of the value arguments.

Valid Syntax	Invalid Syntax

```sprintf('%d', 1, 2, 3, 4)```
```ans =```
    ```1234```
	```sprintf('%1$d', 1, 2, 3, 4)```
	```ans =```
	    ```1```

Also, do not use identifiers when the value arguments are in the form of a vector or array:

Valid Syntax	Invalid Syntax

```v = [1.4 2.7 3.1];```
 	
```sprintf('%.4f %.4f %.4f', v)```
```ans =```
    ```1.4000 2.7000 3.1000```
	```v = [1.4 2.7 3.1];```
	
	```sprintf('%3$.4f %1$.4f %2$.4f', v)```
	```ans =```
	    ```Empty string: 1-by-0```

String Comparisons

There are several ways to compare strings and substrings:

- You can compare two strings, or parts of two strings, for equality.

- You can compare individual characters in two strings for equality.

- You can categorize every element within a string, determining whether each element is a character or white space.

These functions work for both character arrays and cell arrays of strings.

Comparing Strings for Equality

You can use any of four functions to determine if two input strings are identical:

- `strcmp` determines if two strings are identical.

- `strncmp` determines if the first n characters of two strings are identical.

- `strcmpi` and `strncmpi` are the same as `strcmp` and `strncmp`, except that they ignore case.

Consider the two strings

```
str1 = 'hello';
str2 = 'help';
```

Strings `str1` and `str2` are not identical, so invoking `strcmp` returns logical `0` (`false`). For example,

```
C = strcmp(str1,str2)
C =
    0
```

Note For C programmers, this is an important difference between the MATLAB `strcmp` and C `strcmp()` functions, where the latter returns 0 if the two strings are the same.

The first three characters of `str1` and `str2` are identical, so invoking `strncmp` with any value up to 3 returns 1:

```
C = strncmp(str1, str2, 2)
C =
    1
```

These functions work cell-by-cell on a cell array of strings. Consider the two cell arrays of strings

```
A = {'pizza'; 'chips'; 'candy'};
B = {'pizza'; 'chocolate'; 'pretzels'};
```

Now apply the string comparison functions:

```
strcmp(A,B)
ans =
    1
    0
    0
strncmp(A,B,1)
```

```
ans =
     1
     1
     0
```

Comparing for Equality Using Operators

You can use MATLAB relational operators on character arrays, as long as the arrays you are comparing have equal dimensions, or one is a scalar. For example, you can use the equality operator (==) to determine where the matching characters are in two strings:

```
A = 'fate';
B = 'cake';

A == B
ans =
     0    1    0    1
```

All of the relational operators (>, >=, <, <=, ==, ~=) compare the values of corresponding characters.

Categorizing Characters Within a String

There are three functions for categorizing characters inside a string:

1 isletter determines if a character is a letter.

2 isspace determines if a character is white space (blank, tab, or new line).

3 isstrprop checks characters in a string to see if they match a category you specify, such as

- Alphabetic
- Alphanumeric
- Lowercase or uppercase
- Decimal digits
- Hexadecimal digits
- Control characters

- Graphic characters
- Punctuation characters
- Whitespace characters

For example, create a string named `mystring`:

```
mystring = 'Room 401';
```

`isletter` examines each character in the string, producing an output vector of the same length as `mystring`:

```
A = isletter(mystring)
A =
   1   1   1   1   0   0   0   0
```

The first four elements in `A` are logical `1` (`true`) because the first four characters of `mystring` are letters.

Searching and Replacing

MATLAB provides several functions for searching and replacing characters in a string. (MATLAB also supports search and replace operations using regular expressions. See Regular Expressions.)

Consider a string named `label`:

```
label = 'Sample 1, 10/28/95';
```

The `strrep` function performs the standard search-and-replace operation. Use `strrep` to change the date from `'10/28'` to `'10/30'`:

```
newlabel = strrep(label, '28', '30')
newlabel =
   Sample 1, 10/30/95
```

`strfind` returns the starting position of a substring within a longer string. To find all occurrences of the string `'amp'` inside `label`, use

```
position = strfind(label, 'amp')
position =
     2
```

The position within `label` where the only occurrence of `'amp'` begins is the second character.

The `textscan` function parses a string to identify numbers or substrings. Describe each component of the string with conversion specifiers, such as `%s` for strings, `%d` for integers, or `%f` for floating-point numbers. Optionally, include any literal text to ignore.

For example, identify the sample number and date string from `label`:

```
parts = textscan(label, 'Sample %d, %s');
parts{:}

ans =
            1
ans =
    '10/28/95'
```

To parse strings in a cell array, use the `strtok` function. For example:

```
c = {'all in good time'; ...
     'my dog has fleas'; ...
     'leave no stone unturned'};
first_words = strtok(c)
```

Converting from Numeric to String

The functions listed in this table provide a number of ways to convert numeric data to character strings.

Function	Description	Example
char	Convert a positive integer to an equivalent character. (Truncates any fractional parts.)	[72 105] → 'Hi'
int2str	Convert a positive or negative integer to a character type. (Rounds any fractional parts.)	[72 105] → '72 105'
num2str	Convert a numeric type to a character type of the specified precision and format.	[72 105] → '72/105/' (format set to `%1d/`)

Function	Description	Example
mat2str	Convert a numeric type to a character type of the specified precision, returning a string MATLAB can evaluate.	[72 105] → '[72 105]'
dec2hex	Convert a positive integer to a character type of hexadecimal base.	[72 105] → '48 69'
dec2bin	Convert a positive integer to a character type of binary base.	[72 105] → '1001000 1101001'
dec2base	Convert a positive integer to a character type of any base from 2 through 36.	[72 105] → '110 151' (base set to 8)

Converting to a Character Equivalent

The char function converts integers to Unicode character codes and returns a string composed of the equivalent characters:

```
x = [77 65 84 76 65 66];
char(x)
ans =
    MATLAB
```

Converting to a String of Numbers

The int2str, num2str, and mat2str functions convert numeric values to strings where each character represents a separate digit of the input value. The int2str and num2str functions are often useful for labeling plots. For example, the following lines use num2str to prepare automated labels for the x-axis of a plot:

```
function plotlabel(x, y)
plot(x, y)
str1 = num2str(min(x));
str2 = num2str(max(x));
out = ['Value of f from ' str1 ' to ' str2];
xlabel(out);
```

Converting to a Specific Radix

Another class of conversion functions changes numeric values into strings representing a decimal value in another base, such as binary or hexadecimal representation. This includes `dec2hex`, `dec2bin`, and `dec2base`.

Converting from String to Numeric

The functions listed in this table provide a number of ways to convert character strings to numeric data.

Function	Description	Example
`uintN` (e.g., `uint8`)	Convert a character to an integer code that represents that character.	`'Hi'` → 72 105
`str2num`	Convert a character type to a numeric type.	`'72 105'` → [72 105]
`str2double`	Similar to `str2num`, but offers better performance and works with cell arrays of strings.	`{'72' '105'}` → [72 105]
`hex2num`	Convert a numeric type to a character type of specified precision, returning a string that MATLAB can evaluate.	`'A'` → `'-1.4917e-154'`
`hex2dec`	Convert a character type of hexadecimal base to a positive integer.	`'A'` → 10
`bin2dec`	Convert a positive integer to a character type of binary base.	`'1010'` → 10
`base2dec`	Convert a positive integer to a character type of any base from 2 through 36.	`'12'` → 10 (if `base ==` 8)

Converting from a Character Equivalent

Character arrays store each character as a 16-bit numeric value. Use one of the integer conversion functions (e.g., `uint8`) or the `double` function to convert strings to their numeric values, and `char` to revert to character representation:

```
name = 'Thomas R. Lee';

name = double(name)
name =
```

```
       84  104  111  109  97  115  32  82  46  32  76  101  101

name = char(name)
name =
   Thomas R. Lee
```

Converting from a Numeric String

Use str2num to convert a character array to the numeric value represented by that string:

```
str = '37.294e-1';

val = str2num(str)
val =
   3.7294
```

The str2double function converts a cell array of strings to the double-precision values represented by the strings:

```
c = {'37.294e-1'; '-58.375'; '13.796'};

d = str2double(c)
d =
    3.7294
  -58.3750
   13.7960

whos
  Name      Size                    Bytes  Class

  c         3x1                       224  cell
  d         3x1                        24  double
```

Converting from a Specific Radix

To convert from a character representation of a nondecimal number to the value of that number, use one of these functions: hex2num, hex2dec, bin2dec, or base2dec.

The hex2num and hex2dec functions both take hexadecimal (base 16) inputs, but hex2num returns the IEEE double-precision floating-point number it represents, while hex2dec converts to a decimal integer.

Function Summary

MATLAB provides these functions for working with character arrays:

- Functions to Create Character Arrays on page 1-64
- Functions to Modify Character Arrays on page 1-64
- Functions to Read and Operate on Character Arrays on page 1-65
- Functions to Search or Compare Character Arrays on page 1-65
- Functions to Determine Class or Content on page 1-65
- Functions to Convert Between Numeric and String Classes on page 1-66
- Functions to Work with Cell Arrays of Strings as Sets on page 1-66

Functions to Create Character Arrays

Function	Description
'str'	Create the string specified between quotes.
blanks	Create a string of blanks.
sprintf	Write formatted data to a string.
strcat	Concatenate strings.
char	Concatenate strings vertically.

Functions to Modify Character Arrays

Function	Description
deblank	Remove trailing blanks.
lower	Make all letters lowercase.
sort	Sort elements in ascending or descending order.
strjust	Justify a string.

Functions to Modify Character Arrays (Continued)

Function	Description
strrep	Replace one string with another.
strtrim	Remove leading and trailing white space.
upper	Make all letters uppercase.

Functions to Read and Operate on Character Arrays

Function	Description
eval	Execute a string with MATLAB expression.
sscanf	Read a string under format control.

Functions to Search or Compare Character Arrays

Function	Description
regexp	Match regular expression.
strcmp	Compare strings.
strcmpi	Compare strings, ignoring case.
strfind	Find one string within another.
strncmp	Compare the first N characters of strings.
strncmpi	Compare the first N characters, ignoring case.
strtok	Find a token in a string.
textscan	Read data from a string.

Functions to Determine Class or Content

Function	Description
iscellstr	Return true for a cell array of strings.
ischar	Return true for a character array.
isletter	Return true for letters of the alphabet.

Functions to Determine Class or Content (Continued)

Function	Description
isstrprop	Determine if a string is of the specified category.
isspace	Return true for white-space characters.

Functions to Convert Between Numeric and String Classes

Function	Description
char	Convert to a character or string.
cellstr	Convert a character array to a cell array of strings.
double	Convert a string to numeric codes.
int2str	Convert an integer to a string.
mat2str	Convert a matrix to a string you can run eval on.
num2str	Convert a number to a string.
str2num	Convert a string to a number.
str2double	Convert a string to a double-precision value.

Functions to Work with Cell Arrays of Strings as Sets

Function	Description
intersect	Set the intersection of two vectors.
ismember	Detect members of a set.
setdiff	Return the set difference of two vectors.
setxor	Set the exclusive OR of two vectors.
union	Set the union of two vectors.
unique	Set the unique elements of a vector.

Structures

In this section...

What Is a Structure?

A structure is a MATLAB data type that provides the means to store hierarchical data together in a single entity. A structure consists mainly of data containers, called *fields*, and each of these fields stores an array of some MATLAB data type. You assign a name to each field as you create the structure. The figure below shows a structure s that has three fields: a, b, and c.
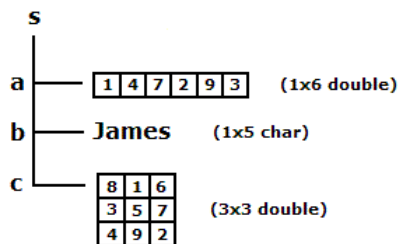
Each field of a structure contains a separate MATLAB array. This array can belong to any one MATLAB or user-defined class, and can have any valid array dimensions. A second field of the same structure can belong to an entirely different class, and can also have different dimensions than the first. The fields of the structure shown above, for example, contain a 1-by-6 array of class `double`, a 1-by-5 array of class `char`, and a 3-by-3 array of `double`.

Like all MATLAB data types, the structure is an array. The class of a structure is called *struct*, so an array of structures is often referred to as a struct array. Like other MATLAB arrays, a struct array can have any dimensions. The struct array shown below has the dimensions 1-by-2 and is composed of two elements: `s(1)` and `s(2)`. Each of these elements is a structure with fields `a`, `b`, and `c` of its own.
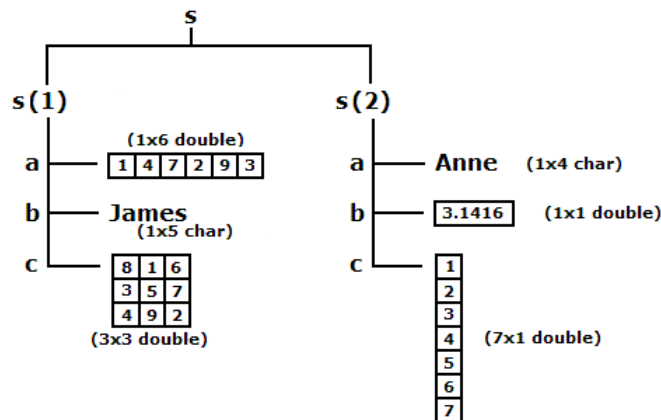
Each element of a struct array must contain the same number of fields and use the same field names as every other element of that struct array. The arrays of data that are stored in these fields, however, do not have to match; they can belong to different classes, and they can have different dimensions. In the struct array shown above, for example, fields `b` and `c` of element `s(1)` contain arrays of different classes and dimensions. The same holds true for fields that are named the same but belong to different elements of the struct array. An example of this is field `b` in `s(1)` and field `b` in `s(2)`.

Reasons to Use a Structure

Perhaps the most common reason for using a structure (or a cell array) is the ability to store arrays of mixed classes and sizes. Most MATLAB arrays must contain the same number of elements which must also be of the same class. The role of the structure, and cell array, as containers of heterogeneous data is very important.

A structure also provides the means to store selected data together in a single entity. This offers you the ability to access and operate on all or parts of the data collectively. You can apply functions directly to a structure, pass the structure to and from your functions, display the value of any fields, or perform most any MATLAB operation on the contents of a structure.

A third reason to use structures is that they give you the ability to apply text labels to your data, as opposed to using array subscripting.

Comparing Struct Arrays with Cell Arrays

Struct arrays and cell arrays are similar in purpose and, in some ways, also design. Both provide a means of storing heterogeneous data in containers of different size and type. Probably the most noticeable *difference* between the two is that the containers of a structure are named fields, whereas a cell array uses numerically indexed cells.

Structures are often used in applications where organization of the data is of high importance. Cell arrays are often used when working with data that you intend to process by index in a programming control loop. Cell arrays are also useful in storing character strings of different lengths.

There are many reasons for choosing either structures, cell arrays, or both for your work in MATLAB. For more information on cell arrays, see "Cell Arrays" on page 1-101 in the MATLAB Programming Fundamentals documentation.

Creating a Structure

This section describes how to create struct arrays of different shapes and sizes, how MATLAB keeps the number of fields the same for all elements, and how to preallocate memory for larger structures.

Creating Structures and Structure Fields

There are two ways of creating a MATLAB structure: by individual assignments to its fields, or by a single call to the `struct` function. The figure below shows a 1-by-1 structure `s` having three fields: `a`, `b`, and `c`. The two sets of commands that can create this structure are shown to the left of and below the figure:

```
s                            s

s.a = 5;          a ──── 5   (1x1 double)

s.b = 10;         b ──── 10  (1x1 double)

s.c = 15;         c ──── 15  (1x1 double)

    OR . . .

s = struct('a', 5, 'b', 10, 'c', 15);
```
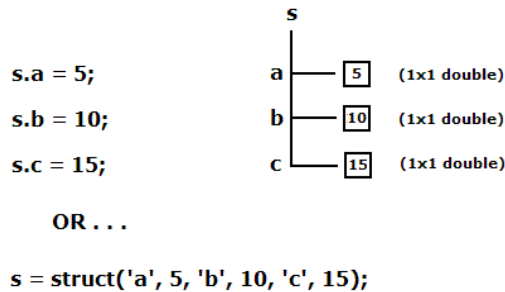
The three statements at the top left are an example of individual assignment to fields. The syntax `s.a` used in the first of these statements refers to field `a` of structure `s`. The structure does not exist yet, so MATLAB creates the structure and the given field `a`, and assigns the value 5 to the field. The two remaining statements add two new fields to the structure and set their values to 10 and 15, respectively.

The statement at the bottom left uses the `struct` function to create the structure and its fields and to assign the respective values to each. See the reference page for the `struct` function for the various syntax options you can use.

Structures with Nonscalar Fields. The next figure creates a three-field structure to contain *nonscalar* array values. The statements used to create this structure are very much like those used in the previous figure.

s.a = [1 4 7 2 9 3];

s.b = 'James';

s.c = magic(3);

OR . . .

s = struct('a', [1 4 7 2 9 3], 'b', 'James', 'c', magic(3));

Nonscalar Struct Arrays. The next figure creates a struct array s that has two elements, s(1) and s(2). Each element of the struct array has three fields: a, b, and c. To create this array by assigning to fields, you need to specify which element of s you are assigning to. For example, to create a field b for element 2 of struct array s, assign to s(2).b.

The struct function requires a slightly different syntax when creating fields in multiple elements of the struct array. Follow each field name argument with a list of values enclosed in curly braces {}. The enclosed list specifies the values to be assigned to that field for the successive elements of the struct array. For example, the first two input arguments shown in the struct command below are 'a' and {[1 4 7 2 9 3],'Anne'}. This tells MATLAB to assign the vector [1 4 7 2 9 3] to s(1).a, and the string 'Anne' to s(2).a:

```
s(1).a = [1 4 7 2 9 3];
s(2).a = 'Anne';

s(1).b = 'James';
s(2).b = pi;

s(1).c = magic(3);
s(2).c = (1:7)';

    OR . . .

s = struct('a', {[1 4 7 2 9 3], 'Anne'},  . . .
           'b', {'James', pi},          . . .
           'c', {magic(3), 1:7});
```
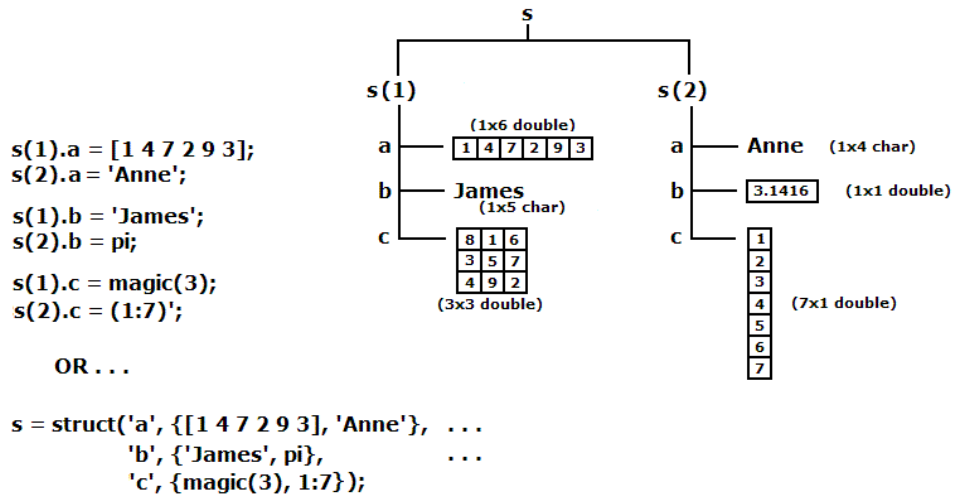
The MATLAB curly braces {} operator constructs a cell array. One use of cell arrays is as a convenient way to pass arguments when calling a function. This is exactly how they are used with the `struct` function. When you use this operator to pass multiple field values to the `struct` function, you are actually passing these values packaged in a cell array. The `struct` function, upon receiving the cell array argument, removes the field values from the cell array and assigns them to the fields specified in your `struct` command.

Suppose that, in the example above, you want to create a field a of for struct element s(1). But instead of s(1).a being a 1-by-6 numeric array, you want it to be a 1-by-6 cell array. In that case, you would need to enclose the first argument itself with curly braces:

```
s = struct('a', {{1 4 7 2 9 3}, 'Anne'}, ...
           'b', {'James', pi}, ...
           'c', {magic(3), (1:7)'});
```

Note When calling the `struct` function, use one set of curly braces {} to pass multiple field values, and use two sets of curly braces {{}} to create a cell array in the specified field.

Nested Structures. The next figure shows an example of one struct array stored (or *nested*) within another struct array. The inner struct array, called myfun, is a collection of two function handles. The commands shown to the left build the two structures, storing the inner structure in field c of the first element of the outer structure:


```
s(1).a = [1 4 7 2 9 3];
s(2).a = 'Anne';

s(1).b = 'James';
s(2).b = pi;

myfun.f1 = @mysqr;
myfun.f2 = @mysum;

s(1).c = myfun;
s(2).c = (1:7)';

    OR . . .


myfun = struct('f1', @mysqr, 'f2', @mysum);
s = struct('a', {[1 4 7 2 9 3], 'Anne'},  . . .
           'b', {'James', pi},           . . .
           'c', {myfun, (1:7)'});
```
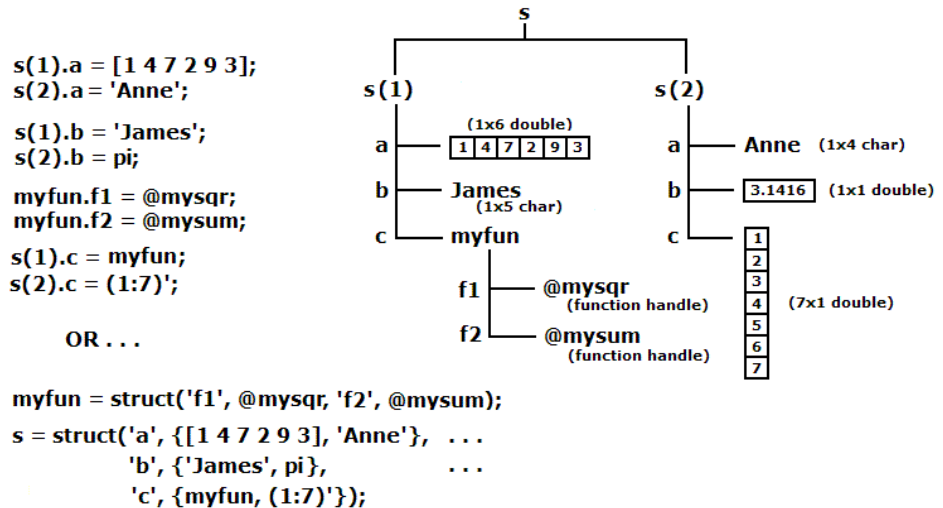
How you organize your structure will depend on your use case. See the reference page for the struct function for more information on using this function to create structures.

Handling Unassigned Fields

Each element of a nonscalar struct array must have the same set of fields. Whenever you add a new field to a struct array, MATLAB adds a field of the same name to all elements of the struct array.

For example, if you enter the following commands:

```
s.a = 5;
s.b = 10;
s(2).a = 15;
```

MATLAB creates and assigns values to the three specified fields, and also creates an unspecified field `s(2).b`, setting its value to the empty array (`[ ]`). This ensures that the fields of `s(1)` and `s(2)` are the same in number and name. This is called *scalar expansion*:

```
s(2).b
ans =
     []
```

MATLAB also automatically keeps field naming and count the same for all elements when you use the `struct` function to create a struct array. In this case, however, field `b` of elements `s(2)` and `s(3)` are set to the value specified for `s(1).b`, which is 10:

```
s = struct('a', {5, 15, 25}, 'b', 10);
[v1 v2 v3] = s.b;

[v1  v2  v3]
ans =
    10    10    10
```

Note The number of field values expressed in curly braces must be the same for each field name with the exception of fields that are scalar, in which case you do not need the curly braces.

This example calls `struct` with three values for field `a` and two values for field `b`, causing the command to generate an error:

```
s = struct('a', {5, 15, 25}, 'b', {10, 15});
??? Error using ==> struct
Array dimensions of input 4 must match those of
   input 2 or be scalar.
```

Preallocating Memory for the Array

MATLAB stores the field names and any overhead information required by a struct array in contiguous memory. If you increase the number of fields used by a struct array over time, even if this happens just by increasing the dimensions of the array, MATLAB uses up more of this contiguous piece of

memory for field name storage. This can eventually lead to "out of memory" errors.

If you can roughly estimate the number of fields and the number of struct array elements at the time you create a struct array, you can preallocate the necessary space in memory and help to avoid this type of problem. See the documentation on Data Structures and Memory to help you make this estimate.

Note Unlike the field name and internal header information of a struct array, the memory consumed by the *data* stored in a struct array is not contiguous, nor can it be preallocated. While preallocating memory can help avoid memory problems when increasing the dimensions of a struct array, it does not protect against shortages in memory due to the amount of data you store in the array. Even with preallocated struct (and cell) arrays, you need to take precautions against using more memory than is available.

How to Preallocate Memory. Methods for preallocating and initializing a struct array are as follows:

- To allocate memory for a0 25-by-50 struct array with fields a, b, and c and initialize the entire array to [ ], use either of the following two methods:

  ```
  S(25,50) = struct( 'a', [], 'b', [], 'c',[]);
  S(25,50).a = [];   S(25,50).b = [];   S(25,50).c = [];
  ```

- To allocate memory for the same struct array, initializing the fields of one element as specified, and copying that element to all elements of the struct array S, use either of the following two methods:

  ```
  S(1:25,1:50) = struct('a', 20, 'b', 30, 'c', 40);
  S = repmat(struct('a', 20, 'b', 30, 'c', 40), [25 50]);
  ```

After the memory has been allocated, you can begin to construct the array by assigning data to it.

Structure Fields

This section describes how to name the fields you create, how to find out what fields a structure contains, how to create and assign field names at run time, and the functions that MATLAB provides to help work with the fields of a structure.

Guidelines for Naming Structure Fields

A field name is just a character string. MATLAB field names must follow the same rules as standard MATLAB variables:

1 Structure field names must begin with a letter, and can contain additional letters, digits, or underscore characters.

2 It is advisable to keep field names to a maximum of N characters, where N is the number returned by the function `namelengthmax`. MATLAB accepts longer names, but only uses the first N characters and ignores the rest.

3 MATLAB distinguishes between uppercase and lowercase characters. The field name `S.income` is not the same as the name `S.Income`.

4 In most cases, you should refrain from using the names of functions or other active variables as field names.

Listing the Fields of a Structure

To access the contents of a struct array, you first need to find out what the names of its fields are. The `fieldnames` function returns a cell array of strings listing all fields belonging to the structure array. The fields appear in the order in which they were created.

Here is a structure with four fields:

```
USPres.name = 'Franklin D. Roosevelt';
USPres.vp(1) = {'John Garner'};
USPres.vp(2) = {'Henry Wallace'};
USPres.vp(3) = {'Harry S. Truman'};
USPres.term = [1933, 1945];
USPres.party = 'Democratic';
```

The `fieldnames` function returns the names of each field of `USPres` in a 4-by-1 cell array of strings:

```
presFields = fieldnames(USPres)
ans =
    'name'
    'vp'
    'term'
    'party'
```

Another means of acquiring fieldnames is to just enter the name of the structure. For a scalar structure, this returns more than just the fieldnames; it also displays the value of each field:

```
USPres =
     name: 'Franklin D. Roosevelt'
       vp: {'John Garner'  'Henry Wallace'  'Harry S. Truman'}
     term: [1933 1945]
    party: 'Democratic'
```

Arranging Fieldnames Alphabetically

By default, MATLAB orders the fieldnames of a structure according to the order in which they were created. The `orderfields` function returns a new struct array that is just like the original, except that the order of the field names is alphabetical. If you assign the output of `orderfields` back to the input structure, it effectively modifies the field ordering of the input structure:

```
USPres = orderfields(USPres);
```

Creating Field Names Dynamically

Another way to give a name to a structure field is to derive the name at the time MATLAB executes your code. First, establish a variable to represent the field name of your structure. Then, at run time, MATLAB uses the current value of this variable as the field name. This is called a *dynamic field name*.

The syntax for creating a field name dynamically is

```
structName.(dynamicExpression) = fieldValue;
```

The term `dynamicExpression` is any MATLAB expression that returns a character or string. For example, in the following statement, the `datestr` function returns the string `Nov2708` which then becomes a field name in the `price` structure. The dot-parentheses `.( )` syntax tells MATLAB that the string value returned by `datestr(now,'mmmddyy')` is a field name for the structure:

```
price.(datestr(now,'mmmddyy')) = 89.99;
```

Examining the field names for the `price` structure shows the `Nov2708` field just added:

```
fieldnames(price)
ans =
    'Nov2708'
```

Functions That Operate on Fields

The following functions are commonly used with the field names of structures. For more information on these functions, consult the MATLAB Function Reference documentation:

Function	Description	Return Value
fieldnames	Get all field names of specified structure.	Cell array of strings listing fields of input structure in the order in which they were assigned to the structure.
getfield	Get contents of the specified field.	Current value assigned to specified field.
isfield	Determine if input is a structure field.	true if the field is a structure field. Otherwise, false.
orderfields	Order fields of a structure array.	Copy of the input structure with fields ordered alphabetically.
rmfield	Remove structure field.	Input structure with specified field removed.
setfield	Set structure field contents, returning the modified structure.	Input structure with specified field set to new value.

To set the value of a structure field, you can either assign it directly, or use the `setfield` function. Likewise, you can obtain the value of a field as shown in the section "Returning Data from a Struct Array" on page 1-82 or by using the `getfield` function.

Concatenating Structures

You can concatenate arrays of structures and arrays of structure fields as explained below.

Concatenating Structure Arrays

When concatenating structure arrays, all of the following must be true:

- All of the arrays must be structures.

- All of the arrays must have the same number of elements along the dimension being joined. For example, you can concatenate a 3-by-5 array of structures with a 9-by-5 array of structures, but you can only do so vertically. See "Keeping Matrices Rectangular".

- All of the arrays must have the same number of fields and the same field names.

- Like-named fields of the arrays being concatenated do *not* have to belong to the same class, or have the same dimensions

Create three struct arrays S1, S2, and S3, each having three fields F1, F2, and F3:

```
rand('state', 0);     % Initialize random number generator
S1 = struct('F1', 'ID_1', 'F2', 2:3:14, 'F3', rand(3,3));
S2 = struct('F1', 'ID_3', 'F2', -2:3:10, 'F3', rand(3,2));
S3 = struct('F1', 'ID_4', 'F2', -3:2:5, 'F3', rand(3,1));
```

Concatenate the three structures to create a 1-by-3 array of structures:

```
 T1 = [S1 S2 S3]
T1 =
1x3 struct array with fields:
    F1
    F2
```

1-79

```
F3
```

This creates a single structure `T1` having the same fields as `S1`, `S2`, and `S3`. Each field in the new structure `T1` contains the data from all of the concatenated fields:

```
T1.F3
ans =
    0.9501    0.4860    0.4565
    0.2311    0.8913    0.0185
    0.6068    0.7621    0.8214
ans =
    0.4447    0.9218
    0.6154    0.7382
    0.7919    0.1763
ans =
    0.4057
    0.9355
    0.9169
```

Concatenating Structure Fields

You concatenate structure field arrays following the same guidelines used when concatenating any other types of arrays.

Indexing into a Struct Array

This section describes how to index into the elements of a struct array, and any arrays that are contained within a struct field.

Basic Struct and Field Indexing

The most general indexing with which to store data into or retrieve data from a struct array is

```
structName(sRows, sCols, ...).fieldName(fRows, fCols, ...)
```

If the structure is scalar, then you can omit the structure indexing as shown here:

```
structName.fieldName(fRows, fCols, ...)
```

Indexing to Inner Levels of the Struct Array

The fields of a structure contain arrays of standard MATLAB data types. These arrays use the indexing syntax appropriate to the class of the array. The table below shows examples of statements that use a combination of struct and cell array indexing:

Array Element	Syntax to Use
Access an element of an array A, where A is a field of structure S.	`S(3,15).A(5,25)`
Access an element of cell array A, where A is a field of structure S.	`S(3,15).A{5,20}`
Access an element of array B, where B is a field of struct array A, and A is a field of struct array S.	`S(3,15).A(5,20).B(50,5)`
Access an element of array B, where B is a field of a structure that resides in cell array A, and A is a field of structure S.	`S(3,15).A{5,20}.B(50,5)`
Access an element of cell array B, where B is a field of structure A, and A is a field of structure S.	`S(3,15).A.B{5,20}`

Indexing Tips

Some techniques that could help you in determining how to format struct array indexing are:

- Use the `whos` function to tell you exactly what the class and size of the variable is that you are dealing with. Combining this information and the standard indexing rules should enable you to find the appropriate syntax to use to get to the desired piece of data.

- Enter only the right side of the assignment statement, effectively assigning to the `ans` variable. By not confining MATLAB to fit its return data into a possibly incompatible data structure, you allow the software to decide the type and size of array needed to contain this data. In so doing, the output illustrates or implies the type of indexing required.

- There are instances in which you can enter a perfectly good indexing statement that will fail just the same. The reason for this failure is that the variable you are attempting to assign to already exists in the workspace. This variable represents an array that is incompatible with your current assignment statement.

  If you are assigning to a variable that is already in use, try clearing the variable from the MATLAB workspace, and then reentering your indexing statement.

- You can index into a nested array in stages rather than all at once. Consider breaking down this indexing expression:

  ```
  S(5,3).A(4,7).B(:,4)
  ```

  into the following:

  ```
  x = S(5,3).A;     % x is a struct array
  y = x(4,7).B;     % y is also a struct array
  z = y(:,4)        % z is a standard array
  ```

Returning Data from a Struct Array

The following table shows a number of different ways of returning data from a struct array. The variable s is a 3-by-4 structure with fields a, b, and c. Each of these fields is a 2-by-5 array of class double:

Values To Be Acquired	MATLAB Statement	Data Structure Returned
The entire struct array s	s	3x4 struct array with fields a, b, and c.
The entire struct array s, as a vector	s(:)	12x1 struct array with fields a, b, and c.
Selected elements of struct s	s(2:3,1:3)	2x3 struct array with fields a, b, and c.
The full array a in selected element of s.	s(2,3).a	2x5 array of double.
The full array a in multiple elements of s.	s(2:3,3:4).a	4-item comma-separated list of 2x5 double.

Values To Be Acquired	MATLAB Statement	Data Structure Returned
The full array `a` in all elements of `s`.	`s.a`	12-item comma-separated list of 2x5 `double`.
Selected elements of `a` in one element of `s`. Multiple elements of `s` are not allowed with this syntax.	`s(3,4).a(2,3:5)`	1x3 array of `double`.

The first three of these indexing expressions provide no access to individual elements of the field arrays. You could use these expressions to copy, rearrange, or delete parts of the structure.

Assigning Struct Array Values to a Comma-Separated List

Accessing a single value from a field in a struct array is no different from accessing one of the elements of any other MATLAB data type. You specify the appropriate subscripts for the struct and field arrays and MATLAB returns the value stored at that location in the array.

Accessing multiple elements, however, can be quite different. Multiple elements of a struct array cannot be assigned to a single variable because they do not necessarily belong to the same class. Instead, MATLAB assigns values from a struct array to a series of separate variables called a comma-separated list.

Create a struct `S`, with one field, `A`:

```
S = struct('A', {5, 'Anne', @myfun, 1:5});
```

Examining field `A` for all elements of `S` returns a comma-separated list:

```
S.A
ans =
     5
ans =
     Anne
ans =
     @myfun
ans =
```

```
     1     2     3     4     5
```

The potential problem with this type of output is that MATLAB overwrites the ans variable for each value returned. If you only want to display these values, then this command should suit your purpose. The next section shows how to assign to variables that you can reuse.

Assigning Struct Values to Separate Variables

If you were to assign multiple elements of a struct array to just one variable, MATLAB uses that variable to return the first value, but is unable to return all values of the array:

```
x = s.a
x =
     5
```

If you know how many values there are in the struct array elements you are trying to access, then you can provide that many outputs in the command, as shown here:

```
[v1 v2 v3 v4] = s.a

v1 =
     5
v2 =
     Anne
v3 =
     @myfun
v4 =
     1     2     3     4     5
```

As in the previous example, this is a comma-separated list. Each return variable is of the class and size of the struct array element assigned to it:

```
whos v1
  Name      Size            Bytes  Class      Attributes

  v1        1x1                 8  double

whos v2
```

```
Name        Size              Bytes  Class    Attributes

v2          1x4                   8  char
```

Assigning Struct Array Values to a Cell Array

You can assign the values of the struct array to a cell array using the
following syntax:

```
[x{1:4}] = s.a
x =
    [5]    'Anne'    @myfun    [1x6 double]

whos x
  Name        Size              Bytes  Class    Attributes

  x           1x4                 320  cell
```

Preallocating the cell array to a certain size and shape gives you control over
how MATLAB returns the output:

```
x1 = cell(4,1);        %  Make x1 a 4-by-1 array
[x1{:}] = s.a
x1 =
    [          5]
    'Anne'
    @myfun
    [1x6 double]

x2 = cell(2,2);        %  Make x2 a 2-by-2 array
[x2{:}] = s.a
x2 =
    [   5]    @myfun
    'Anne'    [1x6 double]

x3 = cell(1,3);        %  Make x3 a 3-element array
[x3{:}] = s.a
x3 =
    [5]    'Anne'    @myfun
```

Another way to do this is to use the following statements:

```
[x1{1:4,1}] = s.a
[x2{1:2,1:2}] = s.a
[x3{1:3}] = s.a
```

Using Structures with Functions

This section describes how to apply a function to data contained within a structure field using the `structfun` function, and also how to pass arguments to and from a function using structures.

Applying a Function to the Fields of a Structure

Use the `structfun` function to execute a function on each field of a scalar struct array. This example executes an anonymous function on a structure having fields that name the days of a week. The anonymous function, `@(x)x(1:3)`, shortens each string to its first three characters. The function reference page for `structfun` explains the use of the `UniformOutput` option:

```
days.f1 = 'Sunday';    days.f2 = 'Monday';
days.f3 = 'Tuesday';   days.f4 = 'Wednesday';
days.f5 = 'Thursday';  days.f6 = 'Friday';
days.f7 = 'Saturday';

shortNames = structfun(@(x)x(1:3), days, 'UniformOutput', false)
shortNames =
    f1: 'Sun'
    f2: 'Mon'
    f3: 'Tue'
    f4: 'Wed'
    f5: 'Thu'
    f6: 'Fri'
    f7: 'Sat'
```

See the reference page for `structfun` for additional help on using this function.

Passing Arguments in a Structure

A simple and easily maintainable way to pass arguments to or from a function is to package them in a structure, and then pass the entire structure to the function. This example passes information pertaining to four United States presidents to the function showPresInfo using only one input argument. You can also use struct arrays to return data from a function call.

Store data on the presidents in a 1-by-30 struct array:

```
USPres(27).name = 'William Howard Taft';   % 27th US President
USPres(27).term = [1909, 1913];            % Term
USPres(27).vp = 'James S. Sherman';        % Vice President

USPres(28).name = 'Woodrow Wilson';        % 28th
USPres(28).term = [1913, 1921];
USPres(28).vp = 'Thomas R. Marshall';

USPres(29).name = 'Warren G. Harding';     % 29th
USPres(29).term = [1921, 1923];
USPres(29).vp = 'Calvin Coolidge';

USPres(30).name = 'Calvin Coolidge';       % 30th
USPres(30).term = [1923, 1929];
USPres(30).vp = 'Charles Dawes';
```

Write a program to display the information passed in:

```
function passFullStructArray(number, info)

% Describe the INFO input.
[dim1,dim2] = size(info);   typ = class(info);
fprintf('\nInput 2 is a %d-by-%d %s array.\n', ...
    dim1, dim2, typ);

% Show the result.
fprintf('\nThe %dth element of the struct contains:\n', number)
info(number)
```

Call the program, passing the entire struct array:

```
passFullStructArray(29, USPres)

Input 2 is a 1-by-30 struct array.

The 29th element of the struct contains:
ans =
    name: 'Warren G. Harding'
    term: [1921 1923]
      vp: 'Calvin Coolidge'
```

Passing Selected Fields in a Structure

You can also pass selected fields of a structure in a function call. This example passes 4 elements of the vp field of the USPres struct array in the form of a comma-separated list. In this case, the function being called, showVPInfo, receives these strings as four separate input arguments:

The value passed to this function is a list of four separate items:

```
USPres(27:30).vp;
ans =
    James S. Sherman
ans =
    Thomas R. Marshall
ans =
    Calvin Coolidge
ans =
    Charles Dawes
```

Write a program that displays the name of a selected Vice President. Use the varargin function to accept and unpack the four separate input arguments generated by the comma-separated list, USPres(27:30).vp:

```
function passPartialStructArray(number, varargin)

% Describe the VARARGIN input.
[dim1,dim2] = size(varargin);    typ = class(varargin);
fprintf('\nInput 2 is a %d-by-%d %s array, VARARGIN.\n', ...
    dim1, dim2, typ);

% Show the result
```

```
str = ['\nThe Vice President who served with ', ...
    'the %dth US President was %s\n'];
n = number - 26;
fprintf(str, number, varargin{n})
```

Call the program, passing a 4-element comma-separated list, which is internally converted to the 1-by-4 VARARGIN cell array:

```
passPartialStructArray(28, USPres(27:30).vp)

Input 2 is a 1-by-4 cell array, VARARGIN.

The Vice President who served with the 28th US President was
    Thomas R. Marshall
```

Converting Between Struct Array and Cell Array

The struct2cell function converts a structure array to a cell array. The statement

```
c = struct2cell(s)
```

converts an m-by-n structure s that has p fields into a p-by-m-by-n cell array c:

The cell2struct function converts a cell array to a struct array. The statement

```
s = cell2struct(c,f,d)
```

converts a cell array c into a struct array s having the fields named in f and based on the d axis of the input cell array.

Conversion Example

This example converts a 1-by-2 struct array USPres_Struct with four fields to a 4-by-1-by-2 cell array USPres_Cell, and then back to a structure USPres_Struct2 that is equal to the original.

Create the original structure:

```
USPresStruct = struct( ...
    'name', ...
```

```
                {'Franklin D. Roosevelt', 'Harry S. Truman'}, ...
            'party', ...
                {'Democratic', 'Democratic'}, ...
            'term', ...
                {{1933, 1945}, {1945, 1953}}, ...
            'vp', ...
                {{'John Garner'; 'Henry Wallace'; 'Harry S. Truman'}, ...
                 {'Alben Barkley'}}));

USPresStruct(1).name = 'Franklin D. Roosevelt';
USPres_s1(1).party = 'Democratic';
USPres_s1(1).term = {1933, 1945};
USPres_s1(1).vp = ...
    {'John Garner';'Henry Wallace';'Harry S. Truman'};


USPresStruct = struct('name', 'Franklin D. Roosevelt', ...
'party', 'Democratic', 'term', {1 2 3}, 'vp', ...
{'John Garner' 'Henry Wallace' 'Harry S. Truman'})


USPres_s1(2).name = 'Harry S. Truman';
USPres_s1(2).party = 'Democratic';
USPres_s1(2).term = {1945, 1953};
USPres_s1(2).vp = 'Alben Barkley';
whos USPres_s1
  Name            Size              Bytes  Class      Attributes

  USPres_s1       1x2                1400  struct
```

Convert the struct array to a cell array:

```
USPresCell = struct2cell(USPresStruct);
whos USPresCell
  Name            Size              Bytes  Class      Attributes

  USPresCell      4x1x2              1208  cell
```

```
USPres_c1 = struct2cell(USPres_s1);
whos USPres_c1
  Name              Size                Bytes  Class     Attributes

  USPres_c1         4x1x2                1144  cell
```

Convert back to a struct array and compare it with the original:

```
USPres_s2 = cell2struct(USPres_c1, ...
    {'name','party','term','vp'}, 1);
whos USPres_s2
  Name              Size                Bytes  Class     Attributes

  USPres_s2         1x2                  1400  struct

isequal(USPresStruct, USPres_Struct2)
ans =
     1
```

Organizing Data in Structure Arrays

The key to organizing structure arrays is to decide how you want to access subsets of the information. This, in turn, determines how you build the array that holds the structures, and how you break up the structure fields.

For example, consider a 128-by-128 RGB image stored in three separate arrays; RED, GREEN, and BLUE.

Blue intensity values

```
0.689 0.706 0.118 0.884 ...
0.535 0.532 0.653 0.925 ...
0.314 0.265 0.159 0.101 ...
0.553 0.633 0.528 0.493 ...
0.441 0.465 0.512 0.512 ...
0.398 0.401 0.421 0.398 ...
                 912 0.713 ...
                 219 0.328 ...
                 128 0.133 ...
```

Green intensity values

```
0.342 0.647 0.515 0.816 ...
0.111 0.300 0.205 0.526 ...
0.523 0.428 0.712 0.929 ...
0.214 0.604 0.918 0.344 ...
0.100 0.121 0.113 0.126 ...
0.288 0.187 0.204 0.175 ...
              760 0.531 ...
              997 0.910 ...
              995 0.726 ...
```

Red intensity values

```
0.112 0.986 0.234 0.432 ...
0.765 0.128 0.863 0.521 ...
1.000 0.985 0.761 0.698 ...
0.455 0.783 0.224 0.395 ...
0.021 0.500 0.311 0.123 ...
1.000 1.000 0.867 0.051 ...
1.000 0.945 0.998 0.893 ...
0.990 0.941 1.000 0.876 ...
0.902 0.867 0.834 0.798 ...
              .
              .
              .
```

There are at least two ways you can organize such data into a structure array: plane organization and element-by-element organization.

Plane organization

Element-by-element organization

1-by-1 structure array where each field is a 128-by-128 array

128-by-128 structure array where each field is a single data element
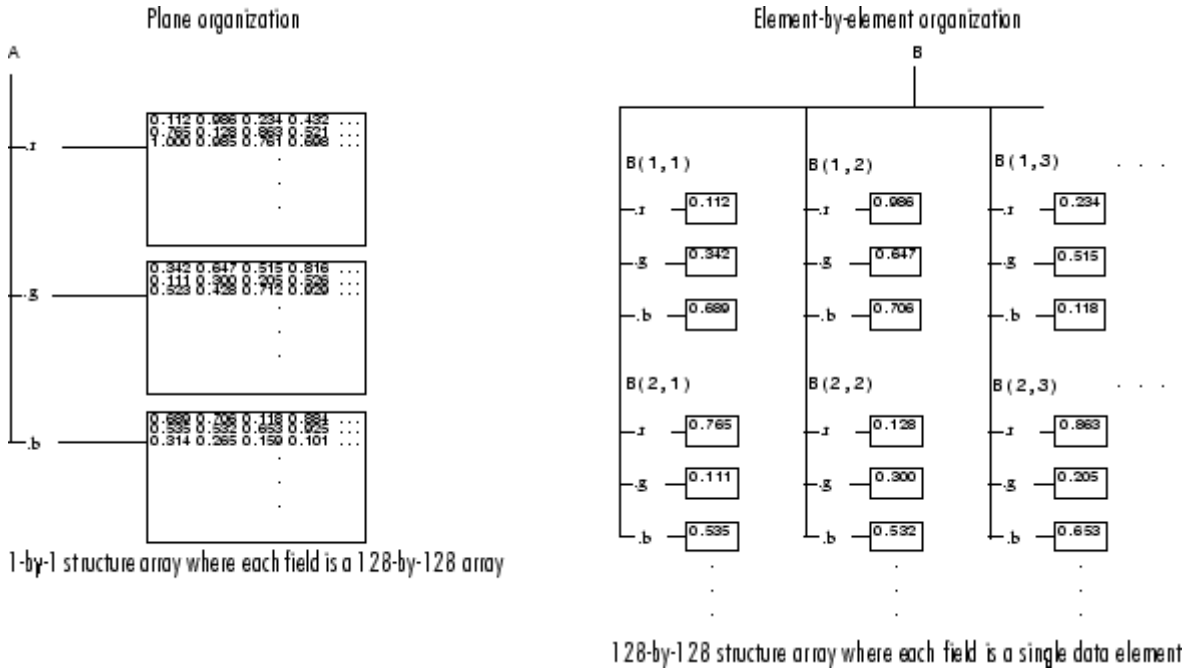
Plane Organization

In the plane organization, shown to the left in the figure above, each field of the structure is an entire plane of the image. You can create this structure using

```
A.r = RED;
A.g = GREEN;
A.b = BLUE;
```

This approach allows you to easily extract entire image planes for display, filtering, or other tasks that work on the entire image at once. To access the entire red plane, for example, use

```
redPlane = A.r;
```

Plane organization has the additional advantage of being extensible to multiple images in this case. If you have a number of images, you can store them as `A(2)`, `A(3)`, and so on, each containing an entire image.

The disadvantage of plane organization is evident when you need to access subsets of the planes. To access a subimage, for example, you need to access each field separately:

```
redSub = A.r(2:12,13:30);
greenSub = A.g(2:12,13:30);
blueSub = A.b(2:12,13:30);
```

Element-by-Element Organization

The element-by-element organization, shown to the right in the previous figure, has the advantage of allowing easy access to subsets of data. However, it has the disadvantage of being very memory-intensive. To set up the data in this organization, use

```
for m = 1:size(RED,1)
   for n = 1:size(RED,2)
      B(m,n).r = RED(m,n);
      B(m,n).g = GREEN(m,n);
      B(m,n).b = BLUE(m,n);
   end
end
```

With element-by-element organization, you can access a subset of data with a single statement:

```
Bsub = B(1:10,1:10);
```

To access an entire plane of the image using the element-by-element method, however, requires a loop:

```
redPlane = zeros(128, 128);
for k = 1:(128 * 128)
    redPlane(k) = B(k).r;
end
```

Element-by-element organization is not the best structure array choice for most image processing applications. However, it can be the best for other applications wherein you routinely need to access corresponding subsets of structure fields. The example in the following section demonstrates this type of application.

Example — A Simple Database

Consider organizing a simple database.
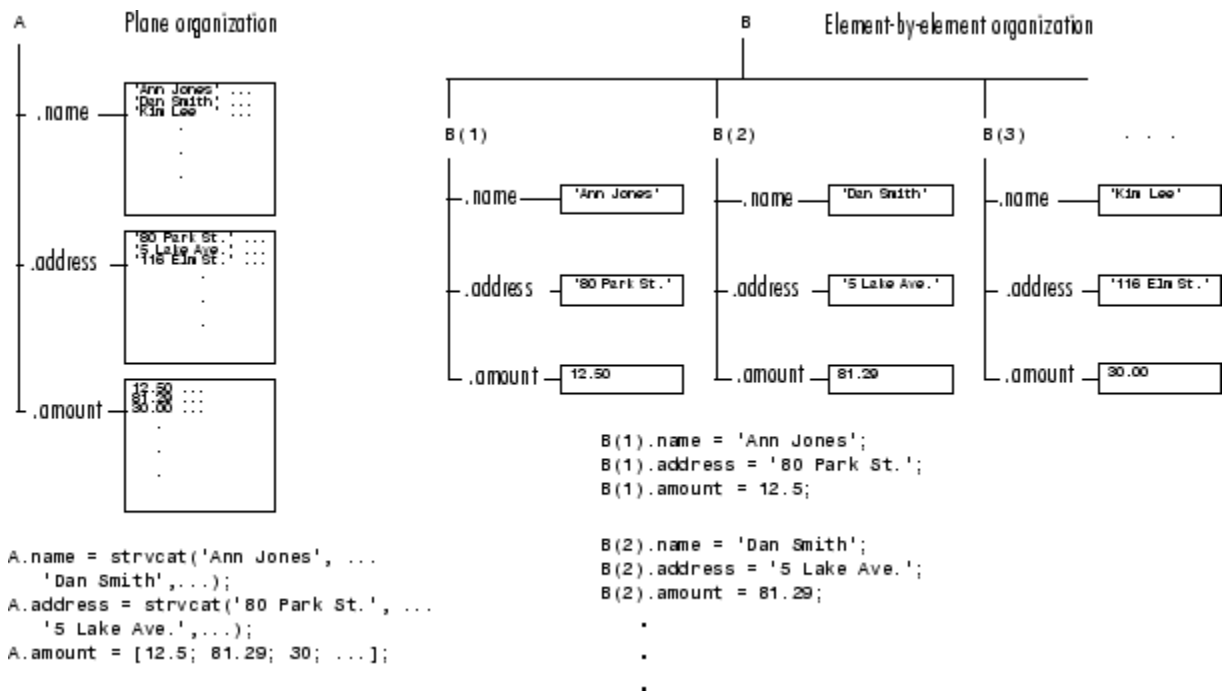
Each of the possible organizations has advantages depending on how you want to access the data. Typically, your data does not dictate the organization scheme you choose. Rather, you must consider how you want to access and operate on the data.

Advantages of Using Plane Organization. Plane organization makes it easier to operate on all field values at once. If, for example, you want to find the average of all the values in the amount field,

1-95

- **Using plane organization**, you could use the following statement:

    ```
    avg = mean(A.amount);
    ```

- **Using element-by-element organization**, you need to remember to enclose the amount expression in brackets:

    ```
    avg = mean([B.amount]);
    ```

Advantages of Using Element-By-Element Organization.

Element-by-element organization makes it easier to access all the information related to a single client. Consider a program file, client.m, which displays the name and address of a given client on screen.

Using plane organization, the client function appears as:

```
function client(name,address)
disp(name)
disp(address)
```

Call this function as follows:

```
client(A.name(2,:), A.address(2,:))
```

Using element-by-element organization, both the function and function call require less code. The client function is:

```
function client(B)
disp(B)
```

To call the function, use:

```
client(B(2))
```

Element-by-element organization also makes it easier to expand the string array fields. If you do not know the maximum string length ahead of time for plane organization, you might need to frequently recreate the name or address field to accommodate longer strings.

Operator Summary

This section summarizes the following types of operators that work with structures:

- "Operators That Construct the Struct Array" on page 1-97
- "Operators That Concatenate Structures" on page 1-97
- "Operators Used for Struct Array Indexing " on page 1-97

Operators That Construct the Struct Array

Syntax	Description
`S = struct('f1', x, 'f2', y, 'f3', z)`	Builds a 1-by-1 struct array `S` with fields `f1`, `f2`, and `f3`, which can contain data of unlike types. Field `f1` contains the value of `x`, field `f2` contains the value of `y`, etc.
`S = struct('f1', {x, y, z})`	Builds a 1-by-3 struct array `S` where each element of `S` has one field `f1`, which can contain data of unlike types. `S(1).f1` contains the value of `x`, `S(2).f1` contains the value of `y`, etc.

Operators That Concatenate Structures

Syntax	Description
`S3 = [S1 S2]`	Concatenates struct arrays `S1` and `S2` into a two-element struct array.
`S3 = [S1.F; S2.F]`	Concatenates the *fields* of struct arrays `S1` and `S2` into an array of the same data type. `S3` is not necessarily a struct.

Operators Used for Struct Array Indexing

Syntax	Description
`X = S(s)`	Returns the elements of struct array `S` specified by subscripts `s`.
`X = S(s).F`	Returns all elements of field `F` for elements of `S` specified by `s`.

Syntax	Description
X = S(s).F(f)	Returns selected elements of field F for structure elements specified by subscript s.
X = S1(s₁).S2(s₂).F	Returns the contents of a nested struct array. Multiple elements of S1 are not allowed with this syntax.
X = S(s).F{c}	Returns the specified elements of a cell array that reside in a field of struct array S.

Function Summary

This section summarizes the following types of functions that work with structures:

- "Functions Related to Constructing the Struct Array" on page 1-98
- "Functions Related to the Type of the Struct Array" on page 1-99
- "Functions Related to Struct Fields" on page 1-99
- "Functions Related to Applying Functions to a Struct Array" on page 1-100

Functions Related to Constructing the Struct Array

Function	Description
cat	Concatenate arrays along specified dimension.
horzcat	Concatenate arrays horizontally.
length	Length of vector.
ndims	Number of array dimensions.
numel	Number of elements in array or subscripted array expression.
repmat	Replicate and tile array.
reshape	Reshape array.
size	Size of array.

Function	Description
struct	Create structure array.
vertcat	Concatenate arrays vertically.

Functions Related to the Type of the Struct Array

Function	Description
cell2struct	Convert cell array to structure array.
class	Create object or return class of object.
isstruct	Determine whether input is structure array.
struct2cell	Convert structure to cell array.
whos	List variables in workspace.

Functions Related to Struct Fields

Function	Description
fieldnames	Get all field names of specified structure.
getfield	Get contents of the specified field.
isempty	Determine whether array is empty.
isfield	Determine if input is a structure field.
orderfields	Order fields of a structure array.
rmfield	Remove structure field.
setfield	Set structure field contents, returning the modified structure.
Dynamic Fieldnames	Generate field name strings at run time.

Functions Related to Applying Functions to a Struct Array

Function	Description
structfun	Apply function to each field of scalar structure.
varargin	Variable length input argument list.
varargout	Variable length output argument list.

Cell Arrays

What Is a Cell Array?

A cell array is a collection of containers called *cells* in which you can store different types of data. The figure shown below represents a 2-by-3 cell array. The cells in row one hold an array of unsigned integers, an array of strings, and an array of complex numbers. Row two holds three other types of arrays, the last being a second cell array nested in the outer one:

cell 1,1	cell 1,2	cell 1,3
3 4 2 9 7 6 8 5 1	'Anne Smith' '9/12/94 ' 'Class II ' 'Obs. 1 ' 'Obs. 2 '	.25+3i 8-16i 34+5i 7+.92i

cell 2,1	cell 2,2	cell 2,3
1.43 2.98 7.83 5.67 4.21	-7 2 -14 8 3 -45 52 -16 3	'text' 4 2 / 1 5 7.3 2.5 / 1.4 0 .02 + 8i

Each cell of a cell array contains some type of MATLAB array. The data in this array can belong to any one MATLAB or user-defined class, and can have any valid array dimensions; this includes 1-by-1 (a scalar array), or having one or more dimension equal to zero (an empty array). A second cell of the same array can belong to an entirely different class, and can also have different dimensions than the first. The capability to store arrays of mixed classes and sizes is the most significant feature of a cell array. Another common use of cell arrays is to store character strings that are of unequal length. A cell array that is used for this purpose is called a *cell array of strings*.

Like all MATLAB arrays, cell arrays must be rectangular in shape. That is, the length of all rows must be the same, the length of all columns the same, and so on for every dimension of the array.

In many respects, cell arrays are quite similar to struct arrays. See "Comparing Struct Arrays with Cell Arrays" on page 1-69 for help in deciding which of these two classes best suits the needs of your applications.

Cell Array Operations

This table shows the operators used in creating, concatenating, and indexing into the cells of a cell array.

Operation	Syntax	Description
Creating	`C = {A B D E}`	Builds a cell array `C` that can contain data of unlike types in `A`, `B`, `D`, and `E`.
Concatenating	`C3 = {C1 C2}`	Concatenates cell arrays `C1` and `C2` into a two-element cell array `C3` such that `C3{1} = C1` and `C3{2} = C2`.
	`C3 = [C1 C2]`	Concatenates the *contents* of cell arrays `C1` and `C2`, assuming that the dimensions of these arrays are compatible.
Indexing	`X = C(s)`	Returns the *cells* of array `C` that are specified by subscripts `s`.
	`X = C{s}`	Returns the *contents* of the cells of `C` that are specified by subscripts `s`.
	`X = C{s}(t)`	References one or more elements of an array that resides within a cell. Subscript `s` selects the cell, and subscript `t` selects the array element(s).

For more information on these operations, see "Creating a Cell Array" on page 1-103, "Concatenating Cell Arrays" on page 1-108, and "Indexing into a Cell Array" on page 1-109, respectively.

Creating a Cell Array

- "Nesting One Cell Array in Another" on page 1-104
- "Creating Cell Arrays One Cell At a Time" on page 1-105
- "Alternative Assignment Syntax" on page 1-107
- "Preallocating Memory for the Array" on page 1-107

Creating cell arrays in MATLAB is similar to creating arrays of other MATLAB classes like double, char, and so on. The main difference is that, when creating a cell array, you enclose the array contents or indices with curly braces { } instead of square brackets [ ]. The curly braces are cell array

constructors, just as square brackets are numeric array constructors. Use commas or spaces to separate elements and semicolons to terminate each row.

For example, to create a 2-by-2 cell array A, type

```
A = {[1 4 3; 0 5 8; 7 2 9], 'Anne Smith'; 3+7i, -pi:pi/4:pi};
```

This results in the array shown below.

Note The curly braces operator creates two-dimensional matrices only, (including 0-by-0, 1-by-1, and 1-by-n matrices). To create cell arrays of more than two dimensions, see "Creating Multidimensional Arrays".

Nesting One Cell Array in Another

To nest one cell array within another, enclose both inner and outer cell arrays with the curly braces { }. The example shown here nests a cell array of vital signs inside a cell array of a person's medical record. (Defining the columns with a header is not usually required, and is just used here to make the example simpler):

```
header = {'Name', 'Age', 'Pulse/Temp/BP'};
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};

header, records
header =
    'Name'    'Age'    'Pulse/Temp/BP'
records =
    'Kelly'    [49]    {1x3 cell}
```

It is often easier to build a nested cell array in steps. The example below creates the inner cell array, vitalsigns, first. The second statement then uses the vitalsigns array in creating the outer cell array, records:

```
vitalsigns = {60, 98.4, [105, 75]};

records(1,:) = {'Kelly', 49, vitalsigns}
record =
    'Name'     'Age'      'Pulse/Temp/BP'
    'Kelly'    [ 49]          {1x3 cell}
```

Verify the new values in the records cell array:

```
fprintf('pulse: %d    temp: %3.1f    bp: %d/%d\n', ...
        records{3}{:})
pulse: 60    temp: 98.4    bp: 105/75
```

Creating Cell Arrays One Cell At a Time

You also can create a cell array one cell at a time by using multiple assignment statements. MATLAB expands the size of the cell array with each assignment statement:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
A(2,1) = {3+7i};
A(2,2) = {-pi:pi/4:pi};
```

If you assign data to a cell that is outside the dimensions of the current array, MATLAB automatically expands the array to include the subscripts you specify. It fills any intervening cells with empty matrices. For example, the assignment below turns the 2-by-2 cell array A into a 3-by-3 cell array:

```
A(3,3) = {5};
```

cell 1,1	cell 1,2	cell 1,3
1 4 3 0 5 8 7 2 9	'Anne Smith'	[ ]
cell 2,1 3+7i	**cell 2,2** [-3.14...3.14]	**cell 2,3** [ ]
cell 3,1 [ ]	**cell 3,2** [ ]	**cell 3,3** 5

3–by-3 Cell Array

Note If you already have a numeric array of a given name, do not try to create a cell array of the same name by assignment without first clearing the numeric array. If you do not clear the numeric array, MATLAB assumes that you are trying to *mix* cell and numeric syntaxes, and generates an error. Similarly, MATLAB does not clear a cell array when you make a single assignment to it. If any of the examples in this section give unexpected results, clear the cell array from the workspace and try again.

Handling Unassigned Cells. To keep all dimensions of a cell array even, MATLAB automatically fills in any unassigned cells as you build the cell array. For example, if you have a cell array that consists of a row of three elements, and you add one new cell to a second row, MATLAB adds two cells to the new row to keep all rows at the same length. The values of these two cells are set to the empty array [ ]. This is called *scalar expansion*.

MATLAB handles other array types in a similar manner, except that it sets unassigned elements to zero instead of the empty array. This example adds a single element to a 1-by-3 array of type `double`, and then does the same to a cell array:

```
A = [2 4 6];   A(2,1) = 8
A =
     2     4     6
```

```
     8      0      0

C = {2 4 6};   C(2,1) = {8}
C =
    [2]    [4]    [6]
    [8]    [ ]    [ ]
```

Alternative Assignment Syntax

When assigning values to a cell array, either of the syntaxes shown below is valid. You can use the braces on the right side of the equation, enclosing the value being assigned, as shown here:

```
A(1,1) = {[1 4 3; 0 5 8; 7 2 9]};
A(1,2) = {'Anne Smith'};
```

You can also use them on the left side, enclosing the array subscripts:

```
A{1,1} = [1 4 3; 0 5 8; 7 2 9];
A{1,2} = 'Anne Smith';
```

Preallocating Memory for the Array

MATLAB stores internal information for a cell array in a contiguous segment of memory called a *header*. If you increase the number of cells in a cell array over time, the size of the header also grows, thus using more of this segment in memory. This can eventually lead to "out of memory" errors.

If you can roughly estimate the dimensions of a cell array at the time you create it, you can preallocate the necessary space in memory and help to avoid this type of problem. See the documentation on Data Structures and Memory to help you make this estimate.

Note Unlike the internal header information of a cell array, the memory consumed by the *data* stored in a cell array is not contiguous, nor can it be be preallocated. While preallocating memory can help avoid memory problems when increasing the dimensions of a cell array, it does not protect against shortages in memory due to the amount of data you store in the array. Even with preallocated cell (and struct) arrays, you need to take precautions against using more memory than is available.

How to Preallocate Memory. To allocate memory for a 25-by-50 cell array and initialize the entire array to [ ], use either of the following two methods:

```
C = cell(25,50);
C{25,50} = [];
```

After the memory has been allocated, you can begin to construct the array by assigning data to it.

Concatenating Cell Arrays

There are two ways that you can create a new cell array from existing cell arrays:

- Concatenate entire cell arrays to individual cells of the new array. For example, join three cell arrays together to build a new cell array having three elements, each containing a cell array. This method uses the curly brace { } operator.

- Concatenate the *contents* of the cells into a new array. For example, join cell arrays of size m-by-n1, m-by-n2, and m-by-n3 together to yield a new cell array that is m-by-(n1+n2+n3) in size. This method uses the square bracket [ ] operator.

Here is an example. First, create three 3-row cell arrays of different widths:

```
C1 = {'Aug' 'Sep';  '10' '17';  uint16(2004) uint16(2004)};
C2 = {'Dec' 'Jan' 'Feb';  '31' '2' '10';  ...
      uint16(2007) uint16(2008) uint16(2008)};
C3 = {'Jun';  '23';  uint16(2002)};
```

This creates arrays C1, C2, and C3:

```
        C1                      C2                      C3
  'Aug'    'Sep'        'Dec'    'Jan'    'Feb'        'Jun'
  '10'     '17'         '31'     '2'      '10'         '23'
  [2004]   [2004]       [2007]   [2008]   [2008]       [2002]
```

Use the curly brace operator to concatenate entire cell arrays, thus building a 1-by-3 cell array from the three initial arrays. Each cell of this new array holds its own cell array:

```
  C4 = {C1 C2 C3}
  C4 =
      {3x2 cell}    {3x3 cell}    {3x1 cell}
```

Now use the square bracket operator on the same combination of cell arrays. This time MATLAB concatenates the contents of the cells together and produces a 3-by-6 cell array:

```
  C5 = [C1 C2 C3]
  C5 =
      'Aug'      'Sep'      'Dec'      'Jan'      'Feb'      'Jun'
      '10'       '17'       '31'       '2'        '10'       '23'
      [2004]     [2004]     [2007]     [2008]     [2008]     [2002]
```

Note The notation {} denotes the empty cell array, just as [ ] denotes the empty matrix for numeric arrays. You can use the empty cell array in any cell array assignments.

Indexing into a Cell Array

When working with cell arrays, you have a choice of selecting entire cells of an array to work with, or the contents of those cells. The first method is *cell indexing*, the second is *content indexing*:

- Cell indexing enables you to work with whole cells of an array. You can access single or multiple cells within the array, but you cannot select anything less than the complete cell. If you want to manipulate the cells of an array without regard to the contents of those cells, use cell indexing. This type of indexing is denoted by the parentheses operator ( ).

Use cell indexing to assign any set of cells to another variable, creating a new cell array.
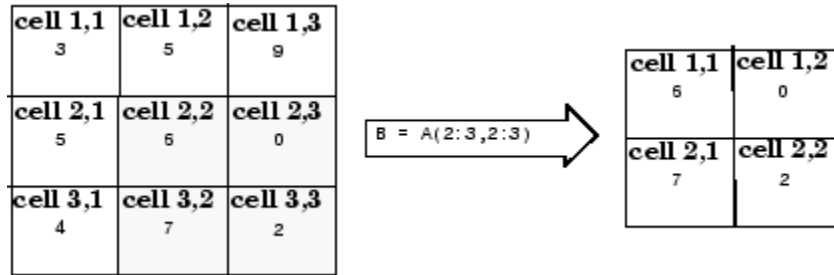
Creating a New Cell Array from an Existing One

- Content indexing gives you access to the contents of a cell. You can work with individual elements of an array within a cell, but you can only do so for one cell at a time. This indexing uses the curly brace operator { }.

Note The examples in this section focus on two-dimensional cell arrays. For examples of higher-dimension cell arrays, see "Multidimensional Arrays".

This example shows how to use cell and content indexing. Start out by creating the following 3-by-3 cell array. The third element of each row is a nested cell array:

```
header = {'Name', 'Age', 'Pulse/Temp/BP'};
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};
records(2,:) = {'Mark', 25, {60, 98.6, [105, 75]}};
records(3,:) = {'Susan', 32, {71, 99.1, [110, 78]}};
```

Display the contents of the cell array. Defining the columns with a `header` is not usually required, and is just used here to make the example simpler:

```
header =
    'Name'    'Age'    'Pulse/Temp/BP'
records =
    'Kelly'    [49]    {1x3 cell}
    'Mark'     [25]    {1x3 cell}
    'Susan'    [32]    {1x3 cell}
```

Use content indexing (curly braces) to change one of the names. Content indexing gives you access to what is contained within the cells of the array:

```
records{3,1}='Susanne'
records =
    'Kelly'      [49]    {1x3 cell}
    'Mark'       [25]    {1x3 cell}
    'Susanne'    [32]    {1x3 cell}
```

Use cell indexing (parentheses) to delete an entire row. (You delete part of a cell array by assigning the empty array [ ] to it.) Cell indexing is appropriate here because you do not need access to the *contents* of the row:

```
records(1,:) = []
records =
    'Mark'       [25]    {1x3 cell}
    'Susanne'    [32]    {1x3 cell}
```

Indexing Into Inner Levels of the Cell Array

The cells of a cell array contain arrays of standard MATLAB data types. These arrays use the indexing syntax appropriate to the class of the array. The table below shows examples of statements that use a combination of cell and struct array indexing:

Action	Required Indexing
Access an element of an array in a cell of cell array C.	C{3,15}(5,25)
Access an element of array A, where A is a field of a structure that resides in cell array C.	C{3,15}.A(5,20)
Access an element of an array that resides in a nested cell array.	C{3,15}{5,20}(50,5)
Access an element of array B, where B is a field of structure A, and A is a field of a structure that resides in cell array C.	C{3,15}.A(5,20).B(50,5)
Access an element of cell array B, where B is a field of a structure that resides in cell array C.	C{3,15}.B{5,20}

Start this example by creating the `records` cell array taken from the example in the previous section:

```
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};
records(2,:) = {'Mark', 25,  {60, 98.6, [105, 75]}};
records(3,:) = {'Susan', 32, {71, 99.1, [110, 78]}};
```

Display information from cells in the nested cell array. This requires two adjacent expressions of content indexing, {2,3}{3}:

```
fprintf('Name: %s   Systolic: %d   Diastolic: %d\n', ...
        records{2,1}, records{2,3}{3})
Name: Mark   Systolic: 105   Diastolic: 75
```

Indexing Tips

You can index into a nested array in stages rather than all at once. Consider breaking down this indexing expression

```
C{5,3}{4,7}(:,4)
```

into the following:

```
x = C{5,3};      % x is a cell array
y = x{4,7};      % y is also a cell array
z = y(:,4)       % z is a standard array
```

See the section on "Indexing Tips" on page 1-81 in the documentation on "Structures" for indexing tips that apply to both the `cell` and `struct` classes.

Using Map Objects in Cell Array Indexing

If you want both the numeric indexing of cell arrays and the named containers of structures, you can combine the two to some extent by implementing cell array indexing with a MATLAB Map object (see "Map Containers" on page 1-150). The Map object provides a translation from a name string to a numeric array index. This implementation has the advantage of using less memory than a struct or cell array, but has the disadvantage of being slower.

The example shown here demonstrates the use of a Map object in locating information in a cell array. Note that the names given to the keys of a Map

object do not have to adhere to the rules for variable names. In this example, each of the key names contain a space character. This is not allowed in variable names:

```
redSoxStats(57:60,1:4) = { ...
%
%   AtBat  Runs  Hits  HR
%
     653,  118,  213,  17;  ... % Pedroia
     554,   98,  155,   9;  ... % Ellsbury
     538,   91,  168,  29;  ... % Youkilis
     423,   37,   93,  13}; ... % Varitek

m1 = containers.Map({'Dustin Pedroia','Jacoby Ellsbury', ...
     'Kevin Youkilis', 'Jason Varitek'}, {57,58,59,60});

m2 = containers.Map({'AtBat','Runs','Hits', 'HR'}, ...
   {1,2,3,4});

player = 'Dustin Pedroia';
fprintf( ...
   '\n   %s had %d At Bats and %d hits this season.\n', ...
   player, redSoxStats{m1(player), m2('AtBat')}, ...
   redSoxStats{m1(player), m2('Hits')})

Dustin Pedroia had 653 At Bats and 213 hits in the 2008 season.
```

Assigning Values to a Cell Array

Use the curly brace { } operator on the right side of the statement to assign values to a cell array:

To store four values in a 2-by-2 cell array, use

```
C = {magic(5), 'Hello'; uint8(100), [1:3:19]}
C =
    [5x5 double]    'Hello'
    [     100]    [1x7 double]
```

The following commands place the values into different cells of cell array C:

```
clear C
C(3,1:4) = {magic(5), 'Hello', uint8(100), [1:3:19]}
C =
                    []            []         []                    []
                    []            []         []                    []
        [5x5 double]        'Hello'      [100]      [1x7 double]


clear C
C(2:3,5:6) = {magic(5), 'Hello'; uint8(100), [1:3:19]}
C =
        []      []      []      []                        []                      []
        []      []      []      []      [5x5 double]      [          100]
        []      []      []      []      'Hello'              [1x7 double]
```

The deal function offers an alternative method of writing to the cell array.
These two statements produce the same result as the statements shown above
that use the curly braces { } operator:

```
[C{3,1:4}] = deal(magic(5), 'Hello', uint8(100), [1:3:19]);
[C{2:3,5:6}] = deal(magic(5), 'Hello', uint8(100), [1:3:19]);
```

Returning Data from a Cell Array

This section describes the syntax to use to have MATLAB return data from a
cell array, how to assign data from a cell array to a comma-separated list or
separate output variables, and also how to plot the contents of a cell array.

Obtaining Values from the Array

The following table shows a number of different ways of returning data from
a cell array. The variable c is a 3x4 cell array in which each cell contains
a 2x5 array of class double.

Values to be acquired	MATLAB Statement	Data Structure Returned
Top level of cell array c	c	3x4 cell array.
Top level of cell array c, as a vector	c(:)	12x1 cell array.
Selected cells in cell array c	c(2:3,1:3)	2x3 cell array.

Values to be acquired	MATLAB Statement	Data Structure Returned
Full contents of one cell in cell array c.	c{2,3}	2x5 array of double.
Full contents of selected cells in cell array c.	c{2:3,3:4}	4-item comma-separated list of 2x5 double.
Full contents of all cells in cell array c.	c{:}	12-item comma-separated list of 2x5 double.
Selected elements of one cell in cell array c. You cannot use multiple elements of c with this syntax.	c{3,4}(2,3:5)	1x3 array of double.

The first three of these indexing expressions provide no access to individual elements of the cells. You could use these expressions to copy, rearrange, or delete parts of the cell array.

Assigning Cell Values to a Comma-Separated List

Accessing a single value from one cell of a cell array is no different from accessing one of the elements of any other MATLAB data type. Accessing multiple elements, however, can be quite different. Multiple elements of a cell array cannot be assigned to a single variable because they do not necessarily belong to the same class. Instead, MATLAB assigns values from a cell array to a series of separate variables called a comma-separated list. Here is an example of such a list:

First, create a 3-by-3 cell array called records:

```
records(1,:) = {'Kelly', 49, {58, 98.3, [103, 72]}};
records(2,:) = {'Mark', 25,  {60, 98.6, [105, 75]}};
records(3,:) = {'Susan', 32, {71, 99.1, [110, 78]}};
```

Displaying one column of the cell array causes MATLAB to return three separate values, each, in succession, assigned to the ans variable:

```
records{:,2}
ans =
    49
ans =
```

```
        25
ans =
        32
```

The potential problem with this type of output is that MATLAB overwrites
the `ans` variable for each value returned. If you only want to display these
values, then this command should suit your purpose. The next section shows
how to assign to variables that you can reuse.

Assigning Cell Values to Separate Variables

If you were to assign multiple elements of a cell array to just one variable,
MATLAB uses that variable to return the first value, but is unable to return
all values of the array:

```
x = records{:,2}
x =
    49
```

If you know how many values there are in the cell array elements you are
trying to access, then you can provide that many outputs in the command, as
shown here:

```
[v1 v2 v3] = records{:,1}
v1 =
    Kelly
v2 =
    Mark
v3 =
    Susan
```

As in the previous example, this is a comma-separated list. As you can see
here, each return variable adopts the class and size of the cell array element
assigned to it:

```
whos v1
  Name      Size            Bytes  Class     Attributes

  v1        1x5                10  char

whos v2
```

```
Name       Size           Bytes  Class    Attributes

v2         1x4               8  char
```

Plotting the Cell Array

For a high-level graphical display of cell architecture, use the `cellplot` function. Consider a 2-by-2 cell array containing two text strings, a matrix, and a vector:

```
c{1,1} = '2-by-2';
c{1,2} = 'eigenvalues of eye(2)';
c{2,1} = eye(2);
c{2,2} = eig(eye(2));
```

The command `cellplot(c)` produces this figure.
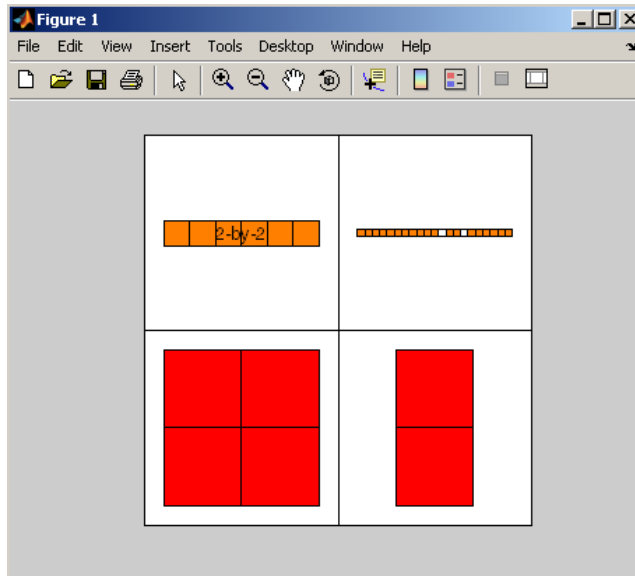
Using Cell Arrays with Functions

This section describes how to apply a function to data contained within a cell array using the `cellfun` function, and also how to pass arguments to and from a function using cell arrays.

Applying a Function to the Cells of a Cell Array

Use the `cellfun` function to run a function on each field of a scalar cell array. This example runs an anonymous function on a cell array containing the days of a week. The anonymous function, `@(x)x(1:3)`, shortens each string to its first three characters. The function reference page for `cellfun` explains the use of the `UniformOutput` option:

```
days{1} = 'Sunday';    days{2} = 'Monday';
days{3} = 'Tuesday';   days{4} = 'Wednesday';
days{5} = 'Thursday';  days{6} = 'Friday';
days{7} = 'Saturday';

shortNames = cellfun(@(x)x(1:3), days, 'UniformOutput', false)
shortNames =
    'Sun'    'Mon'    'Tue'    'Wed'    'Thu'    'Fri'    'Sat'
```

See the reference page for `cellfun` for additional help on using this function.

Passing Variable Numbers of Arguments

You can call a function with variable numbers of input or output arguments by using the terms `varargin` and `varargout` in the respective input and output argument lists for that function. The function being called provides access to these arguments using cell arrays named `varargin` and `varargout`. See Passing Variable Numbers of Arguments in the documentation on "Functions and Scripts".

Passing Arguments in a Cell Array

A simple and easily maintainable way to pass arguments to or from a function is to package them in a cell array, and then pass the entire cell array to the function. This example passes information pertaining to four United States presidents to the function `showPresInfo`:

```
USPres = cell(30,3);    % Allocate memory for the array.
```

```
USPres{27,1} = 'William Howard Taft';    % 27th US President
USPres{27,2} = [1909, 1913];             % Term
USPres{27,3} = 'James S. Sherman';       % Vice President

USPres{28,1} = 'Woodrow Wilson';         % 28th
USPres{28,2} = [1913, 1921];
USPres{28,3} = 'Thomas R. Marshall';

USPres{29,1} = 'Warren G. Harding';      % 29th
USPres{29,2} = [1921, 1923];
USPres{29,3} = 'Calvin Coolidge';

USPres{30,1} = 'Calvin Coolidge';        % 30th
USPres{30,2} = [1923, 1929];
USPres{30,3} = 'Charles Dawes';
```

Write a program to display the information passed in:

```
function passFullCellArray(number, info)

% Describe the INFO input.
[dim1,dim2] = size(info);  typ = class(info);
fprintf('\nInput 2 is a %d-by-%d %s array.\n', ...
    dim1, dim2, typ);

% Show the result.
fprintf('\nThe %dth element of the array contains:\n', ...
    number)
info{number, :}
```

Call the program, passing the entire cell array:

```
passFullCellArray(28, USPres)

Input 2 is a 30-by-3 cell array.

The 28th element of the array contains:
ans =
   Woodrow Wilson
```

```
ans =
        1913        1921
ans =
    Thomas R. Marshall
```

Passing Selected Cells of a Cell Array

You can also pass selected cells of a cell array in a function call. This example passes the names of the four Vice Presidents in the form of a comma-separated list. In this case, the function being called, showVPInfo, receives these strings as four separate input arguments:

The value passed to this function is a list of four separate items:

```
USPres{27:30,3}
ans =
    James S. Sherman
ans =
    Thomas R. Marshall
ans =
    Calvin Coolidge
ans =
    Charles Dawes
```

Write a program that describes the data passed in and then displays the name of a selected Vice President. Use the varargin function to accept and unpack the four separate input arguments generated by the USPres{27:30,3} input:

```
function passPartialCellArray(number, varargin)

% Describe the VARARGIN input.
[dim1, dim2] = size(varargin(:));    typ = class(varargin);
fprintf('\nInput 2 is a %d-by-%d %s array.\n', ...
    dim1, dim2, typ);

% Show the result
str = ['\nThe Vice President who served with ', ...
    'the %dth US President was %s\n'];
n = number - 26;
fprintf(str, number, varargin{n})
```

Call the program, passing a 4-by-1 segment of the cell array:

```
passPartialCellArray(28, USPres{27:30,3})

Input 2 is a 4-by-1 cell array.

The Vice President who served with the 28th US President was
    Thomas R. Marshall
```

Converting Between Cell Array and Struct Array

The `cell2struct` function converts a cell array to a struct array. The statement

```
s = cell2struct(c,f,d)
```

converts a cell array `c` into a struct array `s` having the fields named in `f` and based on the `d` axis of the input cell array.

The `struct2cell` function converts a structure array to a cell array. The statement

```
c = struct2cell(s)
```

converts an `m`-by-`n` structure `s` that has `p` fields into a `p`-by-`m`-by-`n` cell array `c`:

Conversion Example

This example converts a 4-by-1-by-2 cell array `USPres_c1` to a 1-by-2 struct array `USPres_s1` with four fields, and then back to a cell array `USPres_c2` that is equal to the original.

Create the original cell array:

```
USPres_c1{1,1,1} = 'Franklin D. Roosevelt';
USPres_c1{2,1,1} = 'Democratic';
USPres_c1{3,1,1} = [1933, 1945];
USPres_c1{4,1,1} = ...
     {'John Garner';'Henry Wallace';'Harry S. Truman'};

USPres_c1{1,1,2} = 'Harry S. Truman';
USPres_c1{2,1,2} = 'Democratic';
```

```
USPres_c1{3,1,2} = [1945, 1953];
USPres_c1{4,1,2} = {'Alben Barkley'};
whos USPres_c1
  Name            Size              Bytes  Class    Attributes

  USPres_c1       4x1x2               964  cell
```

Convert the cell array to a struct array:

```
USPres_s1 = cell2struct(USPres_c1, ...
    {'name','party','term','vp'}, 1);
whos USPres_s1
  Name            Size              Bytes  Class    Attributes

  USPres_s1       1x2                1220  struct
```

Convert back to a cell array and compare it with the original:

```
USPres_c2 = struct2cell(USPres_s1);
whos USPres_c2
  Name            Size              Bytes  Class    Attributes

  USPres_c2       4x1x2               964  cell

isequal(USPres_c1, USPres_c2)
ans =
     1
```

Operator Summary

This section summarizes the following types of operators that work with cell arrays:

- "Operators That Construct the Cell Array" on page 1-123
- "Operators That Concatenate Cells and Cell Content" on page 1-123
- "Operators Used for Cell Array Indexing" on page 1-123

Operators That Construct the Cell Array

Syntax	Description
C = {A B D E}	Builds a cell array C that can contain data of unlike types in A, B, D, and E.

Operators That Concatenate Cells and Cell Content

Syntax	Description
C3 = {C1 C2}	Concatenates cell arrays C1 and C2 into a two-element cell array C3, such that C3{1} = C1 and C3{2} = C2.
C3 = [C1 C2]	Concatenates the *contents* of cell arrays C1 and C2 into a new cell array with length(C3) == length(C1) + length(C2).

Operators Used for Cell Array Indexing

Syntax	Description
X = C(s)	Returns the *cells* of array C that are specified by subscripts s.
X = C{s}	Returns the *contents* of the cells of C that are specified by subscripts s.
X = C{s}(v)	References one or more elements of an array that resides within a cell. Subscript s selects the single cell, and subscript v selects the array element(s).
X = C{s_1}{s_2}	Returns the contents of a nested cell array. Subscripts for the outer array C are s_1. These subscripts can only refer to one cell of the outer array. Subscripts for the inner cell array are s_2.

Syntax	Description
X = C{s$_1$}{s$_2$}(v)	Returns one or more elements of an array that reside in a nested cell array.
X = C{s}(t).f(v)	Returns one or more elements of an array that reside in a struct field, where the struct resides in a cell of cell array C. Subscripts are s for the cell array, t for the struct array, and v for the lowest-level array.

Function Summary

This section summarizes the following types functions that work with cell arrays:

- "Functions Related to Constructing the Array" on page 1-124
- "Functions Related to the Type of the Array" on page 1-125
- "Functions Related to Obtaining Cell Array Contents" on page 1-125
- "Functions Related to Applying Functions to a Cell Array" on page 1-125
- "Functions Used with Cell Array Conversion" on page 1-126

Functions Related to Constructing the Array

Function	Description
cat	Concatenate arrays along specified dimension.
cell	Create cell array.
horzcat	Concatenate arrays horizontally.
length	Length of array.
ndims	Number of array dimensions.
numel	Number of elements in array or subscripted array expression.
repmat	Replicate and tile array.
reshape	Reshape array.

Function	Description
size	Size of array.
vertcat	Concatenate arrays vertically.

Functions Related to the Type of the Array

Function	Description
cell2struct	Convert cell array to structure array.
class	Create object or return class of object.
iscell	Determine whether input is cell array.
struct2cell	Convert structure to cell array.
whos	List variables in workspace.

Functions Related to Obtaining Cell Array Contents

Function	Description
celldisp	Display cell array contents.
cellplot	Display a graphical depiction of a cell array.
deal	Copy input to separate outputs.

Functions Related to Applying Functions to a Cell Array

Function	Description
cellfun	Apply function to each field of scalar cell array.
varargin	Variable length input argument list.
varargout	Variable length output argument list.

Functions Used with Cell Array Conversion

Function	Description
cell2struct	Convert cell array into struct array.
struct2cell	Convert struct array into cell array.
mat2cell	Divide matrix into cell array of matrices
cell2mat	Convert cell array of matrices to single matrix
num2cell	Convert numeric array to cell array

Function Handles

In this section...
"Overview" on page 1-127
"Creating a Function Handle" on page 1-127
"Calling a Function By Means of Its Handle" on page 1-131
"Preserving Data from the Workspace " on page 1-133
"Applications of Function Handles" on page 1-136
"Saving and Loading Function Handles" on page 1-142
"Advanced Operations on Function Handles" on page 1-142
"Functions That Operate on Function Handles" on page 1-149

Overview

A *function handle* is a callable association to a MATLAB function. It contains an association to that function that enables you to invoke the function regardless of where you call it from. This means that, even if you are outside the normal scope of a function, you can still call it if you use its handle.

With function handles, you can:

- Pass a function to another function
- Capture data values for later use by a function
- Call functions outside of their normal scope
- Save the handle in a MAT-file to be used in a later MATLAB session

See "Applications of Function Handles" on page 1-136 for an explanation of each of these applications.

Creating a Function Handle

- "Maximum Length of a Function Name" on page 1-128

- "The Role of Scope, Precedence, and Overloading When Creating a Function Handle" on page 1-129
- "Obtaining Permissions from Class Methods" on page 1-129
- "Using Function Handles for Anonymous Functions" on page 1-130
- "Arrays of Function Handles" on page 1-131

You construct a handle for a specific function by preceding the function name with an @ sign. The syntax is:

```
h = @functionname
```

where `h` is the variable to which the returned function handle is assigned.

Use only the function *name*, with no path information, after the @ sign. If there is more than one function with this name, MATLAB associates with the handle the one function source it would dispatch to if you were actually calling the function.

Create a handle `h` for a function `plot` that is on your MATLAB path:

```
h = @plot;
```

Once you create a handle for a function, you can invoke the function by means of the handle instead of using the function name. Because the handle contains the absolute path to its function, you can invoke the function from any location that MATLAB is able to reach, as long as the program file for the function still exists at this location. This means that functions in one file can call functions that are not on the MATLAB path, subfunctions in a separate file, or even functions that are private to another folder, and thus not normally accessible to that caller.

Maximum Length of a Function Name

Function names used in handles are unique up to `N` characters, where `N` is the number returned by the function `namelengthmax`. If the function name exceeds that length, MATLAB truncates the latter part of the name.

For function handles created for Sun™ Java™ constructors, the length of any segment of the package name or class name must not exceed `namelengthmax`

characters. (The term *segment* refers to any portion of the name that lies before, between, or after a dot. For example, `java.lang.String` has three segments). The overall length of the string specifying the package and class has no limit.

The Role of Scope, Precedence, and Overloading When Creating a Function Handle

At the time you create a function handle, MATLAB must decide exactly which function it is to associate the handle to. In doing so, MATLAB uses the same rules used to determine which file to invoke when you make a function call. To make this determination, MATLAB considers the following:

- Scope — The function named must be on the MATLAB path at the time the handle is constructed.

- Precedence — MATLAB selects which function(s) to associate the handle with, according to the function precedence rules described under Determining Which Function Gets Called.

- Overloading — If additional files on the path overload the function for any of the standard MATLAB classes, such as `double` or `char`, then MATLAB associates the handle with these files, as well.

Program files that overload a function for classes other than the standard MATLAB classes are not associated with the function handle at the time it is constructed. Function handles do operate on these types of overloaded functions, but MATLAB determines which implementation to call at the time of evaluation in this case.

Obtaining Permissions from Class Methods

When creating a function handle inside a method of a class, the function is resolved using the permissions of that method. When MATLAB invokes the function handle, it does so using the permissions of the class. This gives MATLAB the same access as the location where the function handle was created, including access to private and protected methods accessible to that class.

Example. This example defines two methods. One, updateObj, defines a listener for an event called Update, and the other , callbackfcn, responds to this event whenever it should occur. The latter function is a private function and thus would not normally be within the scope of the notify function. However, because @callbackfcn is actually a function handle, it retains the permissions of the context that created the function handle:

```
classdef updateObj < handle
   events
      Update
   end

   methods
      function obj = updateObj(varargin)
         addlistener(obj, 'Update', @callbackfcn);
         notify(obj, 'Update');
      end
   end

   methods (Access = private)
      function obj = callbackfcn(obj, varargin)
         disp('Object Updated')
         disp(obj);
      end
   end
end
```

To run this function, invoke updateObj at the MATLAB command line.

Using Function Handles for Anonymous Functions

Function handles also serve as the means of invoking anonymous functions. An anonymous function is a one-line expression-based MATLAB function that does not require a program file.

For example, the statement

```
sqr = @(x) x.^2;
```

creates an anonymous function that computes the square of its input argument x. The @ operator makes sqr a function handle, giving you a means of calling the function:

```
sqr(20)
ans =
    400
```

Like nested functions, a handle to an anonymous function also stores all data that will be needed to resolve the handle when calling the function. Shares same issues as nested functions do.

See the documentation on "Anonymous Functions" on page 4-3 for more information.

Arrays of Function Handles

To create an array of function handles, you must use a cell array:

```
trigFun = {@sin, @cos, @tan};
```

For example, to plot the cosine of the range -pi to pi at 0.01 intervals, use

```
plot(trigFun{2}(-pi:0.01:pi))
```

Calling a Function By Means of Its Handle

Function handles can give you access to functions you might not be able to execute. For instance, with a handle you can call a function even if it is no longer on your MATLAB path. You can also call a subfunction from outside of the file that defines that function.

Calling Syntax

The syntax for calling a function using a function handle is the same used when calling the function directly. For example, if you call function myFun like this:

```
[out1, out2, ...] = myFun(in1, in2, ...);
```

then you would call it using a handle in the same way, but using the handle name instead:

```
fHandle = @myFun;
[out1, out2, ...] = fHandle(in1, in2, ...);
```

There is one small difference. If the function being called takes no input arguments, then you must call the function with empty parentheses placed after the handle name. If you use only the handle name, MATLAB just identifies the name of the function:

```
% This identifies the handle.        % This invokes the function.

fHandle = @computer;                  fHandle = @computer;
fHandle                               fHandle()
ans =                                 ans =
   @computer                             PCWIN
```

Calling a Function with Multiple Outputs

The example below returns multiple values from a call to an anonymous function. Create anonymous function f that locates the nonzero elements of an array, and returns the row, column, and value of each element in variables row, col, and val:

```
f = @(X)find(X);
```

Call the function on matrix m using the function handle f. Because the function uses the MATLAB find function which returns up to three outputs, you can specify from 0 to 3 outputs in the call:

```
m = [3 2 0; -5 0 7; 0 0 1]
m =
     3     2     0
    -5     0     7
     0     0     1

[row col val] = f(m);

val
val =
     3
    -5
```

```
2
7
1
```

Returning a Handle for Use Outside of a Function File

As stated previously, you can use function handles to call a function that may otherwise be hidden or out of scope. This example function `getHandle` returns a function handle `fHandle` to a caller that is outside of the file:

```
function fHandle = getHandle
fHandle = @subFun;

function res = subFun(t1, t2, varargin);
...
```

Call `getHandle` to obtain a function handle with which to invoke the subfunction. You can now call the subfunction as you would any function that is in scope:

```
f1 = getHandle;
result = f1(startTime, endTime, procedure);
```

Example — Using Function Handles in Optimization

Function handles can be particularly useful in optimization work. If you have the MathWorks Optimization Toolbox installed, click on any of the links below to find information and examples on this topic:

- "Passing Extra Parameters" — Calling objective or constraint functions that have parameters in addition to the independent variable.

- "Anonymous Function Objectives" — Use function handles in writing simple objective functions.

- "Example: Nonlinear Curve Fitting with lsqcurvefit" — An example using `lsqcurvefit`, which takes two inputs for the objective.

Preserving Data from the Workspace

Both anonymous functions and nested functions make use of variable data that is stored outside the body of the function itself. For example, the

anonymous function shown here uses two variables: X and K. You pass the X variable into the anonymous function whenever you invoke the function. The value for K however is taken from the currently active workspace:

```
K = 200;
fAnon = @(X)K * X;

fAnon([2.54 1.43 0.68 1.90 1.02 2.13]);
```

What would happen if you tried to invoke this function after you cleared K from the workspace? Or if you saved the anonymous function to a .mat file and then loaded it into an entirely separate computing environment where K is not defined?

The answer is that MATLAB stores any values needed by an anonymous (or nested) function within the handle itself. It does this at the time you construct the handle. This does not include values from the argument list as these values get passed in whenever you call the function.

Preserving Data with Anonymous Functions

If you create an anonymous function at the MATLAB command window, that function has access to the workspace of your current MATLAB session. If you create the function inside of another function, then it has access to the outer function's workspace. Either way, if your anonymous function depends upon variables from an outside workspace, then MATLAB stores the variables and their values within the function handle at the time the handle is created.

This example puts a 3-by-5 matrix into the base workspace, and then creates a function handle to an anonymous function that requires access to the matrix.

Create matrix A and anonymous function testAnon:

```
A = magic(5);  A(4:5,:) = []
A =
    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22

testAnon = @(x)x * A;          % Anonymous function
```

Call the anonymous function via its handle, passing in a multiplier value. This multiplies all elements by 5.2:

```
testAnon(5.2)
ans =
    88.4000   124.8000     5.2000    41.6000    78.0000
   119.6000    26.0000    36.4000    72.8000    83.2000
    20.8000    31.2000    67.6000   104.0000   114.4000
```

Clear variable A from the base workspace and verify that this action has no effect on the output of the anonymous function:

```
clear A

testAnon(5.2)
ans =
    88.4000   124.8000     5.2000    41.6000    78.0000
   119.6000    26.0000    36.4000    72.8000    83.2000
    20.8000    31.2000    67.6000   104.0000   114.4000
```

This works because the variable A and its value at the time the anonymous function is created is preserved within the function handle that also provides access to the anonymous function. See "Variables Used in the Expression" on page 4-8 for more information.

Preserving Data with Nested Functions

Nested functions are similar to anonymous functions in their ability to preserve workspace data needed by a function handle. In this case however, the workspace belongs to one of the functions inside of which the handle is being created. See "Variable Scope in Nested Functions" on page 4-19 and "Using Function Handles with Nested Functions" on page 4-21 for more information on this subject.

This example shows a function getHandles that returns a handle to nested function getApproxVal_V4. The nested function uses two variables, const and adjust, from the workspace of the outer function. Calling getHandles creates a function handle to the nested function and also stores these two variables within that handle so that they will be available whenever the nested function is invoked:

```
function handle = getHandles(adjust)
const = 16.3;
handle = @getApproxVal_V4;

   function vOut = getApproxVal_V4(vectIn)
   vOut = ((vectIn+adjust)*const) + ((vectIn-adjust)*const);
   end
end
```

Call the getHandles function to obtain a handle to the nested function:

```
adjustVal = 0.023;
getApproxValue = getHandles(adjustVal);

getApproxValue([0.67 -0.09 1.01 0.33 -0.14 -0.23])
ans =
   21.8420   -2.9340   32.9260   10.7580   -4.5640   -7.4980
```

The documentation on "Examining a Function Handle" on page 1-143 explains how to see which variables are stored within a particular function handle. Another helpful resource is "Using Function Handles with Nested Functions" on page 4-21.

Loading a Saved Handle to a Nested Function. If you save a function handle to a nested function and, at some later date, modify the function and then reload the handle, you may observe unexpected behavior from the restored handle. when you invoke the function from the reloaded handle.

Applications of Function Handles

The following sections discuss the advantages of using function handles:

- "Pass a Function to Another Function" on page 1-137

- "Capture Data Values For Later Use By a Function" on page 1-138

- "Call Functions Outside of Their Normal Scope" on page 1-141

- "Save the Handle in a MAT-File for Use in a Later MATLAB Session" on page 1-142

Example of Passing a Function Handle

The following example creates a handle for a function supplied by MATLAB called humps and assigns it to the variable h. (The humps function returns a strong maxima near x = 0.3 and x = 0.9).

```
h = @humps;
```

After constructing the handle, you can pass it in the argument list of a call to some other function, as shown here. This example passes the function handle h that was just created as the first argument in a call to fminbnd. This function then minimizes over the interval [0.3, 1].

```
x = fminbnd(h, 0.3, 1)
x =
    0.6370
```

Using a function handle enables you to pass different functions for fminbnd to use in determining its final result.

Pass a Function to Another Function

The ability to pass variables to a function enables you to run the function on different values. In the same way, you can pass function handles as input arguments to a function, thus allowing the called function to change the operations it runs on the input data.

Example 1 — Run quad on Varying Functions. Run the quadrature function on varying input functions:

```
a = 0;  b = 5;

quad(@log, a, b)
ans =
    3.0472

quad(@sin, a, b)
ans =
    0.7163

quad(@humps, a, b)
ans =
```

```
    12.3566
```

Example 2 — Run quad on Anonymous Functions. Run quad on a
MATLAB built-in function or an anonymous function:

```
n = quad(@log, 0, 3);

n = quad(@(x)x.^2, 0, 3);
```

Change the parameters of the function you pass to quad with a simple
modification of the anonymous function that is associated with the function
handle input:

```
a = 3.7;
z = quad(@(x)x.^a, 0, 3);
```

Example 3 — Compare quad Results on Different Functions. Compare
the integral of the cosine function over the interval [a, b]:

```
a = 0; b = 10;
int1 = quad(@cos,a,b)

int1 =
    -0.5440
```

with the integral over the same interval of the piecewise polynomial pp that
approximates the cosine function by interpolating the computed values x
and y:

```
x = a:b;
y = cos(x);
pp = spline(x,y);
int2 = quad(@(x)ppval(pp,x), a, b)

int2 =
    -0.5485
```

Capture Data Values For Later Use By a Function

You can do more with a function handle than just create an association to a
certain function. By using anonymous functions, you can also capture certain

variables and their values from the function workspace and store them in the handle. These data values are stored in the handle at the time of its construction, and are contained within the handle for as long as it exists. Whenever you then invoke the function by means of its handle, MATLAB supplies the function with all variable inputs specified in the argument list of the function call, and also any constant inputs that were stored in the function handle at the time of its construction.

Storing some or all input data in a function handle enables you to reliably use the same set of data with that function regardless of where or when you invoke the handle. You can also interrupt your use of a function and resume it with the same data at a later time simply by saving the function handle to a MAT-file.

Example 1 — Constructing a Function Handle that Preserves Its Variables. Compare the following two ways of implementing a simple plotting function called draw_plot. The first case creates the function as one that you would call by name and that accepts seven inputs specifying coordinate and property information:

```
function draw_plot(x, y, lnSpec, lnWidth, mkEdge, mkFace, mkSize)
plot(x, y, lnSpec, ...
        'LineWidth', lnWidth, ...
        'MarkerEdgeColor', mkEdge, ...
        'MarkerFaceColor', mkFace, ...
        'MarkerSize', mkSize)
```

The second case implements draw_plot as an anonymous function to be called by a function handle, h. The draw_plot function has only two inputs now; the remaining five are specified only on a call to the handle constructor function, get_plot_handle:

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, ...
   mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
```

```
            end
    end
```

Because these input values are required by the draw_plot function but are not made available in its argument list, MATLAB supplies them by storing them in the function handle for draw_plot at the time it is constructed. Construct the function handle h, also supplying the values to be stored in handle:

```
    h = get_plot_handle('--rs', 2, 'k', 'g', 10);
```

Now call the function, specifying only the x and y inputs:

```
    x = -pi:pi/10:pi;
    y = tan(sin(x)) - sin(tan(x));
    h(x, y)                            % Draw the plot
```

The later section on "Examining a Function Handle" on page 1-143 continues this example by showing how you can examine the contents of the function and workspace contents of this function handle.

Example 2 — Varying Data Values Stored in a Function Handle.
Values stored within a handle to a nested function do not have to remain constant. The following function constructs and returns a function handle h to the anonymous function addOne. In addition to associating the handle with addOne, MATLAB also stores the initial value of x in the function handle:

```
    function h = counter
    x = 0;
    h = @addOne;
        function y = addOne;
        x = x + 1;
        y = x;
        end
    end
```

The addOne function that is associated with handle h increments variable x each time you call it. This modifies the value of the variable stored in the function handle:

```
    h = counter;
    h()
```

```
ans =
     1
h()
ans =
     2
```

Example 3 — You Cannot Vary Data in a Handle to an Anonymous Function. Unlike the example above, values stored within a handle to an anonymous function do remain constant. Construct a handle to an anonymous function that just returns the value of x, and initialize x to 300. The value of x within the function handle remains constant regardless of how you modify x external to the handle:

```
x = 300;
h = @()x;

x = 50;
h()
ans =
   300

clear x
h()
ans =
   300
```

Call Functions Outside of Their Normal Scope

By design, only functions within a program file are permitted to access subfunctions defined within that file. However, if, in this same file, you were to construct a function handle for one of the internal subfunctions, and then pass that handle to a variable that exists outside of the file, access to that subfunction would be essentially unlimited. By capturing the access to the subfunction in a function handle, and then making that handle available to functions external to the file (or to the command line), the example extends that scope. An example of this is shown in the preceding section, "Capture Data Values For Later Use By a Function" on page 1-138.

Private functions also have specific access rules that limit their availability with the MATLAB environment. But, as with subfunctions, MATLAB allows

you to construct a handle for a private function. Therefore, you can call it by means of that handle from any location or even from the MATLAB command line, should it be necessary.

Save the Handle in a MAT-File for Use in a Later MATLAB Session

If you have one or more function handles that you would like to reuse in a later MATLAB session, you can store them in a MAT-file using the `save` function and then use `load` later on to restore them to your MATLAB workspace.

Saving and Loading Function Handles

You can save and load function handles in a MAT-file using the MATLAB `save` and `load` functions. If you load a function handle that you saved in an earlier MATLAB session, the following conditions could cause unexpected behavior:

- Any of the files that define the function have been moved, and thus no longer exist on the path stored in the handle.

- You load the function handle into an environment different from that in which it was saved. For example, the source for the function either does not exist or is located in a different folder than on the system on which the handle was saved.

In both of these cases, the function handle is now invalid because it is no longer associated with any existing function code. Although the handle is invalid, MATLAB still performs the load successfully and without displaying a warning. Attempting to invoke the handle, however, results in an error.

Invalid or Obsolete Function Handles

If you create a handle to a function that is not on the MATLAB path, or if you `load` a handle to a function that is no longer on the path, MATLAB catches the error only when the handle is invoked. You can assign an invalid handle and use it in such operations as `func2str`. MATLAB catches and reports an error only when you attempt to use it in a runtime operation.

Advanced Operations on Function Handles

Advanced operations include:

Examining a Function Handle

Use the `functions` function to examine the contents of a function handle.

Caution MATLAB provides the `functions` function for querying and debugging purposes only. Because its behavior may change in subsequent releases, you should not rely upon it for programming purposes.

The following example is a continuation of an example in an earlier section of the Function Handles documentation. See Example 1 in the section "Capture Data Values For Later Use By a Function" on page 1-138 for the complete example.

Construct a function handle that contains both a function association, and data required by that function to execute. The following function constructs the function handle, h:

```
function h = get_plot_handle(lnSpec, lnWidth, mkEdge, ...
   mkFace, mkSize)
h = @draw_plot;
    function draw_plot(x, y)
        plot(x, y, lnSpec, ...
            'LineWidth', lnWidth, ...
            'MarkerEdgeColor', mkEdge, ...
            'MarkerFaceColor', mkFace, ...
            'MarkerSize', mkSize)
    end
end
```

Use `functions` to examine the contents of the returned handle:

```
f = functions(h)
f =
     function: 'get_plot_handle/draw_plot'
```

```
           type: 'nested'
           file: 'D:\matlab\work\get_plot_handle.m'
      workspace: {[1x1 struct]}
```

The call to `functions` returns a structure with four fields:

- `function` — Name of the function or subfunction to which the handle is associated. (Function names that follow a slash character (/) are implemented in the program code as subfunctions.)

- `type` — Type of function (e.g., simple, nested, anonymous)

- `file` — Filename and path to the file. (For built-in functions, this is the string `'MATLAB built-in function'`)

- `workspace` — Variables in the function workspace at the time the handle was constructed, along with their values

Examine the workspace variables that you saved in the function handle:

```
f.workspace{:}
ans =
            h: @get_plot_handle/draw_plot
       lnSpec: '--rs'
      lnWidth: 2
     mrkrEdge: 'k'
     mrkrFace: 'g'
     mrkrSize: 10
```

Converting to and from a String

Two functions, `str2func` and `func2str` enable you to convert between a string containing a function name and a function handle that is associated with that function name.

Converting a String to a Function Handle. Another means of creating a function handle is to convert a string that holds a function name to a handle for the named function. You can do this using the `str2func` function:

```
handle = str2func('functionname');
```

The example below takes the name of a function as the first argument. It compares part of the name to see if this is a polynomial function, converts the function string to a function handle if it is not, and then calls the function by means of its handle:

```
function run_function(funcname, arg1, arg2)
if strncmp(funcname, 'poly', 4)
    disp 'You cannot run polynomial functions on this data.'
    return
else
   h = str2func(funcname);
   h(arg1, arg2);
end
```

Note Nested functions are not accessible to str2func. To construct a function handle for a nested function, you must use the function handle constructor, @.

Converting a Function Handle to a String. You can also convert a function handle back into a string using the func2str function:

```
functionname = func2str(handle);
```

This example converts the function handle h to a string containing the function name, and then uses the function name in a message displayed to the user:

```
function call_h(h, arg1, arg2)
sprintf('Calling function %s ...\n', func2str(h))
h(arg1, arg2)
```

Comparing Function Handles

This section describes how MATLAB determines whether or not separate function handles are equal to each other:

- "Comparing Handles Constructed from a Named Function" on page 1-146
- "Comparing Handles to Anonymous Functions" on page 1-146

1-145

Comparing Handles Constructed from a Named Function. MATLAB considers function handles that you construct from the same named function (e.g., handle = @sin) to be equal. The isequal function returns a value of true when comparing these types of handles:

```
func1 = @sin;
func2 = @sin;
isequal(func1, func2)
 ans =
      1
```

If you save these handles to a MAT-file, and then load them back into the workspace later on, they are still equal.

Comparing Handles to Anonymous Functions. Unlike handles to named functions, any two function handles that represent the same anonymous function (i.e., handles to anonymous functions that contain the same text) are not equal. This is because MATLAB cannot guarantee that the frozen values of non-argument variables (such as A, below) are the same.

```
A = 5;
h1 = @(x)A * x.^2;
h2 = @(x)A * x.^2;

isequal(h1, h2)
ans =
     0
```

Note In general, MATLAB may underestimate the equality of function handles. That is, a test for equality may return false even when the functions happen to behave the same. But in cases where MATLAB does indicate equality, the functions are guaranteed to behave in an identical manner.

If you make a copy of an anonymous function handle, the copy and the original are equal:

```
h1 = @(x)A * x.^2;   h2 = h1;
isequal(h1, h2)
ans =
     1
```

Comparing Handles to Nested Functions. MATLAB considers function handles to the same nested function to be equal only if your code constructs these handles on the same call to the function containing the nested functions. Given this function that constructs two handles to the same nested function:

```
function [h1, h2] = test_eq(a, b, c)
h1 = @findZ;
h2 = @findZ;

    function z = findZ
    z = a.^3 + b.^2 + c';
    end
end
```

function handles constructed from the same nested function and on the same call to the parent function are considered equal:

```
[h1 h2] = test_eq(4, 19, -7);

isequal(h1, h2),
ans =
     1
```

while those constructed from different calls are not considered equal:

```
[q1 q2] = test_eq(3, -1, 2);

isequal(h1, q1)
ans =
     0
```

Comparing Handles Saved to a MAT-File. If you save equivalent anonymous or nested function handles to separate MAT-files, and then load them back into the MATLAB workspace, they are no longer equal. This is because saving the function handle loses track of the original circumstances under which the function handle was created. Reloading it results in a function handle that compares as being unequal to the original function handle.

Create two equivalent anonymous function handles:

```
h1 = @(x) sin(x);
h2 = h1;

isequal(h1, h2)
ans =
     1
```

Save each to a different MAT-file:

```
save fname1 h1;
save fname2 h2;
```

Clear the MATLAB workspace, and then load the function handles back into the workspace:

```
clear all
load fname1
load fname2
```

The function handles are no longer considered equal:

```
isequal(h1, h2)
ans =
     0
```

Note, however, that equal anonymous and nested function handles that you save to the same MAT-file are equal when loaded back into MATLAB.

Functions That Operate on Function Handles

MATLAB provides the following functions for working with function handles.
See the reference pages for these functions for more information.

Function	Description
functions	Return information describing a function handle.
func2str	Construct a function name string from a function handle.
str2func	Construct a function handle from a function name string.
save	Save a function handle from the current workspace to a MAT-file.
load	Load a function handle from a MAT-file into the current workspace.
isa	Determine if a variable contains a function handle.
isequal	Determine if two function handles are handles to the same function.

Map Containers

In this section...
"Overview of the Map Data Structure" on page 1-150
"Description of the Map Class" on page 1-151
"Creating a Map Object" on page 1-153
"Examining the Contents of the Map" on page 1-156
"Reading and Writing Using a Key Index" on page 1-157
"Modifying Keys and Values in the Map" on page 1-160
"Mapping to Different Value Types" on page 1-163

Overview of the Map Data Structure

A *Map* is a type of fast key lookup data structure that offers a flexible means of indexing into its individual elements. Unlike most array data structures in the MATLAB software that only allow access to the elements by means of integer indices, indices for a Map can be nearly any scalar numeric value or a character string.

Indices into the elements of a Map are called *keys*. These keys, along with the data *values* associated with them, are stored within the Map. Each entry of a Map contains exactly one unique key and its corresponding value. Indexing into the Map of rainfall statistics shown below with a string representing the month of August yields the value internally associated with that month, 37.3.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

Aug ────────────▶ (Aug → 37.3) ────────────▶ 37.3

Mean monthly rainfall statistics (mm)

Keys are not restricted to integers as they are with other arrays. Specifically, a key may be any of the following types:

- 1-by-N character array

- Scalar real `double` or `single`

- Signed or unsigned scalar integer

The values stored in a Map can be of any type. This includes arrays of numeric values, structures, cells, strings, objects, or other Maps.

Note A Map is most memory efficient when the data stored in it is a scalar number or a character array.

Description of the Map Class

A Map is actually an object, or instance, of a MATLAB class called `Map`. It is also a handle object and, as such, it behaves like any other MATLAB handle object. This section gives a brief overview of the `Map` class. For more details,

see the function reference pages for the `Map` constructor or for any method of the class.

Properties of the Map Class

All objects of the `Map` class have three properties. You cannot write directly to any of these properties; you can change them only by means of the methods of the `Map` class.

Property	Description	Default
Count	Unsigned 64-bit integer that represents the total number of key/value pairs contained in the `Map` object.	0
KeyType	String that indicates the type of all keys contained in the `Map` object. `KeyType` can be any of the following: `double`, `single`, `char`, and signed or unsigned 32-bit or 64-bit integer. If you attempt to add keys of an unsupported type, `int8` for example, MATLAB makes them `double`.	char
ValueType	String that indicates the type of values contained in the `Map` object. If the values in a Map are all scalar numbers of the same type, `ValueType` is set to that type. If the values are all character arrays, `ValueType` is `'char'`. Otherwise, `ValueType` is `'any'`.	any

To examine one of these properties, follow the name of the Map object with a dot and then the property name. For example, to see what type of keys are used in Map `mapObj`, use

```
mapObj.KeyType
```

A Map is a handle object. As such, if you make a copy of the object, MATLAB does not create a new Map; it creates a new handle for the existing Map that you specify. If you alter the Map's contents in reference to this new handle, MATLAB applies the changes you make to the original Map as well. You can, however, delete the new handle without affecting the original Map.

Methods of the Map Class

The Map class implements the following methods. Their use is explained in the later sections of this documentation and also in the function reference pages.

Method	Description
isKey	Check if Map contains specified key
keys	Names of all keys in Map
length	Length of Map
remove	Remove key and its value from Map
size	Dimensions of Map
values	Values contained in Map

Creating a Map Object

A Map is an object of the Map class. It is defined within a MATLAB package called containers. As with any class, you use its constructor function to create any new instances of it. You must include the package name when calling the constructor:

```
newMap = containers.Map(optional_keys_and_values)
```

Constructing an Empty Map Object

When you call the Map constructor with no input arguments, MATLAB constructs an empty Map object. When you do not end the command with a semicolon, MATLAB displays the following information about the object you have constructed:

```
newMap = containers.Map()
newMap =
  containers.Map handle
  Package: containers

  Properties:
        Count: 0
      KeyType: 'char'
    ValueType: 'any'
```

```
Methods, Events, Superclasses
```

The properties of an empty `Map` object are set to their default values:

- `Count = 0`
- `KeyType = 'char'`
- `ValueType = 'any'`

Once you construct the empty Map object, you can use the `keys` and `values` methods to populate it. For a summary of MATLAB functions you can use with a Map object, see "Methods of the Map Class" on page 1-153

Constructing An Initialized Map Object

Most of the time, you will want to initialize the Map with at least some keys and values at the time you construct it. You can enter one or more sets of keys and values using the syntax shown here. The brace operators (`{}`) are not required if you enter only one key/value pair:

```
mapObj = containers.Map({key1, key2, ...}, {val1, val2, ...});
```

For those keys and values that are character strings, be sure that you specify them enclosed within single quotation marks. For example, when constructing a Map that has character string keys, use

```
mapObj = containers.Map(...
    {'keystr1', 'keystr2', ...}, {val1, val2, ...});
```

As an example of constructing an initialized `Map` object, create a new Map for the following key/value pairs taken from the monthly rainfall map shown earlier in this section.

KEYS	VALUES
Jan	327.2
Feb	368.2
Mar	197.6
Apr	178.4
May	100.0
Jun	69.9
Jul	32.3
Aug	37.3
Sep	19.0
Oct	37.0
Nov	73.2
Dec	110.9
Annual	1551.0

```
k = {'Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', ...
  'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 'Annual'};

v = {327.2, 368.2, 197.6, 178.4, 100.0,  69.9, ...
  32.3,  37.3,  19.0,  37.0,  73.2, 110.9, 1551.0};

rainfallMap = containers.Map(k, v)
rainfallMap =
  containers.Map handle
  Package: containers

  Properties:
        Count: 13
      KeyType: 'char'
    ValueType: 'double'
  Methods, Events, Superclasses
```

The Count property is now set to the number of key/value pairs in the Map, 13, the KeyType is char, and the ValueType is double.

Combining Map Objects

You can combine Map objects vertically using concatenation. However, the result is not a vector of Maps, but rather a single Map object containing all key/value pairs of the contributing Maps. Horizontal vectors of Maps are not allowed. See "Building a Map with Concatenation" on page 1-159, below.

Examining the Contents of the Map

Each entry in a Map consists of two parts: a unique key and its corresponding value. To find all the keys in a Map, use the keys method. To find all of the values, use the values method.

Create a new Map called tickets that maps airline ticket numbers to the holders of those tickets. Construct the Map with four key/value pairs:

```
ticketMap = containers.Map(...
    {'2R175', 'B7398', 'A479GY', 'NZ1452'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham', ...
     'Bradley Reid'});
```

Use the keys method to display all keys in the Map. MATLAB lists keys of type char in alphabetical order, and keys of any numeric type in numerical order:

```
keys(ticketMap)
ans =
    '2R175'   'A479GY'   'B7398'   'NZ1452'
```

Next, display the values that are associated with those keys in the Map. The order of the values is determined by the order of the keys associated with them.

This table shows the keys listed in alphabetical order:

keys	values
2R175	James Enright
A479GY	Sarah Latham

keys	values
B7398	Carl Haynes
NZ1452	Bradley Reid

The `values` method uses the same ordering of values:

```
values(ticketMap)
ans =
   'James Enright'  'Sarah Latham'  'Carl Haynes'  'Bradley Reid'
```

Reading and Writing Using a Key Index

When reading from the Map, use the same keys that you have defined and associated with particular values. Writing new entries to the Map requires that you supply the values to store with a key for each one .

Note For a large Map, the keys and value methods use a lot of memory as their outputs are cell arrays.

Reading From the Map

After you have constructed and populated your Map, you can begin to use it to store and retrieve data. You use a Map in the same manner that you would an array, except that you are not restricted to using integer indices. The general syntax for looking up a value (`valueN`) for a given key (`keyN`) is shown here. If the key is a character string, enclose it in single quotation marks:

```
valueN = mapObj(keyN);
```

You can find any single value by indexing into the map with the appropriate key:

```
passenger = ticketMap('2R175')
passenger =
   James Enright
```

Find the person who holds ticket `A479GY`:

```
sprintf('   Would passenger %s please come to the desk?\n', ...
    ticketMap('A479GY'))
ans =
    Would passenger Sarah Latham please come to the desk?
```

To access the values of multiple keys, use the `values` method, specifying the keys in a cell array:

```
values(ticketMap, {'2R175', 'B7398'})
ans =
    'James Enright'    'Carl Haynes'
```

Map containers support scalar indexing only. You cannot use the colon operator to access a range of keys as you can with other MATLAB classes. For example, the following statements throw an error:

```
ticketMap('2R175':'B7398')
ticketMap(:)
```

Adding Key/Value Pairs

Unlike other array types, each entry in a Map consists of two items: the value and its key. When you write a new value to a Map, you must supply its key as well. This key must be consistent in type with any other keys in the Map.

Use the following syntax to insert additional elements into a Map:

```
existingMapObj(newKeyName) = newValue;
```

Add two more entries to the `ticketMap` used in the above examples, Verify that the Map now has five key/value pairs:

```
ticketMap('947F4') = 'Susan Spera';
ticketMap('417R93') = 'Patricia Hughes';

ticketMap.Count
ans =
     6
```

List all of the keys and values in Map `ticketMap`:

```
keys(ticketMap),  values(ticketMap)
ans =
    '2R175'    '417R93'    '947F4'    'A479GY'    'B7398'    'NZ1452'
ans =
  Columns 1 through 3
    'James Enright'    'Patricia Hughes'    'Susan Spera'
  Columns 4 through 6
    'Sarah Latham'    'Carl Haynes'    'Bradley Reid'
```

Building a Map with Concatenation

You can add key/value pairs to a Map in groups using concatenation. The
concatenation of Map objects is different from other classes. Instead of
building a vector of s, MATLAB returns a single Map containing the key/value
pairs from each of the contributing Map objects.

Rules for the concatenation of Map objects are:

• Only vertical vectors of Map objects are allowed. You cannot create an
  m-by-n array or a horizontal vector of s. For this reason, vertcat is
  supported for Map objects, but not horzcat.

• All keys in each map being concatenated must be of the same class.

• You can combine Maps with different numbers of key/value pairs. The
  result is a single Map object containing key/value pairs from each of the
  contributing maps:

```
tMap1 = containers.Map({'2R175', 'B7398', 'A479GY'}, ...
    {'James Enright', 'Carl Haynes', 'Sarah Latham'});

tMap2 = containers.Map({'417R93', 'NZ1452', '947F4'}, ...
    {'Patricia Hughes', 'Bradley Reid', 'Susan Spera'});

% Concatenate the two maps:
ticketMap = [tMap1; tMap2];
```

The result of this concatenation is the same 6-element map that was
constructed in the previous section:

```
ticketMap.Count
```

```
ans =
    6

keys(ticketMap), values(ticketMap)
ans =
    '2R175'   '417R93'   '947F4'   'A479GY'   'B7398'   'NZ1452'
ans =
  Columns 1 through 3
    'James Enright'      'Patricia Hughes'      'Susan Spera'
  Columns 4 through 6
    'Sarah Latham'      'Carl Haynes'      'Bradley Reid'
```

• Concatenation does not include duplicate keys or their values in the
  resulting Map object.

  In the following example, both objects m1 and m2 use a key of 8. In Map m1,
  8 is a key to value C; in m2, it is a key to value X:

  ```
  m1 = containers.Map({1, 5, 8}, {'A', 'B', 'C'});
  m2 = containers.Map({8, 9, 6}, {'X', 'Y', 'Z'});
  ```

  Combine m1 and m2 to form a new Map object, m:

  ```
  m = [m1; m2];
  ```

  The resulting Map object m has only five key/value pairs. The value C was
  dropped from the concatenation because its key was not unique:

  ```
  keys(m), values(m)
  ans =
      [1]    [5]    [6]    [8]    [9]
  ans =
      'A'    'B'    'Z'    'X'    'Y'
  ```

Modifying Keys and Values in the Map

In addition to reading and writing the contents of a Map, you can also delete
key/value pairs and modify any of its values or keys.

Note Keep in mind that if you have more than one handle to a Map, modifying the handle also makes changes to the original Map. See "Modifying a Copy of the Map" on page 1-162, below.

Removing Keys and Values from the Map

Use the remove method to delete any entries from a Map. When calling this method, specify the Map object name and the key name to remove. MATLAB deletes the key and its associated value from the Map.

The syntax for the remove method is

```
remove('mapName', 'keyname');
```

Remove one entry (the specified key and its value) from the Map object:

```
remove(ticketMap, 'NZ1452');
values(ticketMap)
ans =
  Columns 1 through 3
    'James Enright'    'Patricia Hughes'    'Susan Spera'
  Columns 4 through 5
    'Sarah Latham'    'Carl Haynes'
```

Modifying Values

You can modify any value in a Map simply by overwriting the current value. The passenger holding ticket A479GY is identified as Sarah Latham:

```
ticketMap('A479GY')
ans =
    Sarah Latham
```

Change the passenger's first name to Anna Latham by overwriting the original value for the A479GY key:

```
ticketMap('A479GY') = 'Anna Latham';
```

Verify the change:

```
ticketMap('A479GY')
ans =
    'Anna Latham';
```

Modifying Keys

To modify an existing key while keeping the value the same, first remove both the key and its value from the Map. Then create a new entry, this time with the corrected key name.

Modify the ticket number belonging to passenger James Enright:

```
remove(ticketMap, '2R175');
ticketMap('2S185') = 'James Enright';

k = keys(ticketMap);  v = values(ticketMap);
str1 = '   ''%s'' has been assigned a new\n';
str2 = '     ticket number: %s.\n';

fprintf(str1, v{1})
fprintf(str2, k{1})

  'James Enright' has been assigned a new
     ticket number: 2S185.
```

Modifying a Copy of the Map

Because ticketMap is a handle object, you need to be careful when making copies of the Map. Keep in mind that by copying a Map object, you are really just creating another handle to the same object. Any changes you make to this handle are also applied to the original Map.

Make a copy of Map ticketMap. Write to this copy, and notice that the change is applied to the original Map object itself:

```
copiedMap = ticketMap;
```

```
copiedMap('AZ12345') = 'unidentified person';
ticketMap('AZ12345')
ans =
    unidentified person
```

Clean up:

```
remove(ticketMap, 'AZ12345');
clear copiedMap;
```

Mapping to Different Value Types

It is fairly common to store other classes, such as structures or cell arrays, in a Map structure. However, Maps are most memory efficient when the data stored in them belongs to one of the basic MATLAB types such as double, char, integers, and logicals.

Mapping to a Structure Array

The following example maps airline seat numbers to structures that contain information on who occupies the seat. To start out, create the following structure array:

```
s1.ticketNum = '2S185'; s1.destination = 'Barbados';
s1.reserved = '06-May-2008'; s1.origin = 'La Guardia';
s2.ticketNum = '947F4'; s2.destination = 'St. John';
s2.reserved = '14-Apr-2008'; s2.origin = 'Oakland';
s3.ticketNum = 'A479GY'; s3.destination = 'St. Lucia';
s3.reserved = '28-Mar-2008'; s3.origin = 'JFK';
s4.ticketNum = 'B7398'; s4.destination = 'Granada';
s4.reserved = '30-Apr-2008'; s4.origin = 'JFK';
s5.ticketNum = 'NZ1452'; s5.destination = 'Aruba';
s5.reserved = '01-May-2008'; s5.origin = 'Denver';
```

Map five of the seats to one of these structures:

```
seatingMap = containers.Map( ...
    {'23F', '15C', '15B', '09C', '12D'}, ...
    {s5, s1, s3, s4, s2});
```

Using this Map object, find information about the passenger, who has reserved seat 09C:

```
seatingMap('09C')
ans =
      ticketNum: 'B7398'
    destination: 'Granada'
       reserved: '30-Apr-2008'
         origin: 'JFK'

seatingMap('15B').ticketNum
ans =
    A479GY
```

Using two Maps together, you can find out the name of the person who has reserved the seat:

```
passenger = ticketMap(seatingMap('15B').ticketNum)
passenger =
    Anna Latham
```

Mapping to a Cell Array

As with structures, you can also map to a cell array in a Map object. Continuing with the airline example of the previous sections, some of the passengers on the flight have "frequent flyer" accounts with the airline. Map the names of these passengers to records of the number of miles they have used and the number of miles they still have available:

```
accountMap = containers.Map( ...
    {'Susan Spera','Carl Haynes','Anna Latham'}, ...
    {{247.5, 56.1}, {0, 1342.9}, {24.6, 314.7}});
```

Use the Map to retrieve account information on the passengers:

```
name = 'Carl Haynes';
acct = accountMap(name);

fprintf('%s has used %.1f miles on his/her account,\n', ...
    name, acct{1})
fprintf('  and has %.1f miles remaining.\n', acct{2})

Carl Haynes has used 0.0 miles on his/her account,
```

```
and has 1342.9 miles remaining.
```

Combining Unlike Classes

In this section...
"Combining Unlike Integer Types" on page 1-167
"Combining Integer and Noninteger Data" on page 1-169
"Combining Cell Arrays with Non-Cell Arrays" on page 1-169
"Empty Matrices" on page 1-169
"Concatenation Examples" on page 1-170

Matrices and arrays can be composed of elements of most any MATLAB data type as long as all elements in the matrix are of the same type. If you do include elements of unlike classes when constructing a matrix, MATLAB converts some elements so that all elements of the resulting matrix are of the same type. (See Chapter 1, "Classes (Data Types)" for information on any of the MATLAB classes discussed here.)

Data type conversion is done with respect to a preset precedence of classes. The following table shows the five classes you can concatenate with an unlike type without generating an error (that is, with the exception of character and logical).

TYPE	character	integer	single	double	logical
character	character	character	character	character	invalid
integer	character	integer	integer	integer	integer
single	character	integer	single	single	single
double	character	integer	single	double	double
logical	invalid	integer	single	double	logical

For example, concatenating a double and single matrix always yields a matrix of type single. MATLAB converts the double element to single to accomplish this.

Combining Unlike Integer Types

If you combine different integer types in a matrix (e.g., signed with unsigned, or 8-bit integers with 16-bit integers), MATLAB returns a matrix in which all elements are of one common type. MATLAB sets all elements of the resulting matrix to the data type of the left-most element in the input matrix. For example, the result of the following concatenation is a vector of three 16-bit signed integers:

```
A = [int16(450) uint8(250) int32(1000000)]
```

MATLAB also displays a warning to inform you that the result may not be what you had expected:

```
A = [int16(450) uint8(250) int32(1000000)];
Warning: Concatenation with dominant (left-most) integer class
may overflow other operands on conversion to return class.
```

You can disable this warning by entering the following two commands directly after the operation that caused the warning. The first command retrieves the message identifier associated with the most recent warning issued by MATLAB. The second command uses this identifier to disable any further warnings of that type from being issued:

```
[msg, intcat_msgid] = lastwarn;
warning('off', intcat_msgid);
```

To reenable the warning so that it will now be displayed, use

```
warning('on', intcat_msgid);
```

You can use these commands to disable or enable the display of any MATLAB warning.

Example of Combining Unlike Integer Sizes

After disabling the integer concatenation warnings as shown above, concatenate the following two numbers once, and then switch their order. The return value depends on the order in which the integers are concatenated. The left-most type determines the data type for all elements in the vector:

```
A = [int16(5000) int8(50)]
```

```
A =
   5000    50

B = [int8(50) int16(5000)]
B =
   50   127
```

The first operation returns a vector of 16-bit integers. The second returns a vector of 8-bit integers. The element int16(5000) is set to 127, the maximum value for an 8-bit signed integer.

The same rules apply to vertical concatenation:

```
C = [int8(50); int16(5000)]
C =
    50
   127
```

Note You can find the maximum or minimum values for any MATLAB integer type using the intmax and intmin functions. For floating-point types, use realmax and realmin.

Example of Combining Signed with Unsigned

Now do the same exercise with signed and unsigned integers. Again, the left-most element determines the data type for all elements in the resulting matrix:

```
A = [int8(-100) uint8(100)]
A =
   -100    100

B = [uint8(100) int8(-100)]
B =
   100    0
```

The element int8(-100) is set to zero because it is no longer signed.

MATLAB evaluates each element *prior to* concatenating them into a combined array. In other words, the following statement evaluates to an 8-bit signed integer (equal to 50) and an 8-bit unsigned integer (unsigned -50 is set to zero) before the two elements are combined. Following the concatenation, the second element retains its zero value but takes on the unsigned `int8` type:

```
A = [int8(50), uint8(-50)]
A =
  50    0
```

Combining Integer and Noninteger Data

If you combine integers with `double`, `single`, or `logical` classes, all elements of the resulting matrix are given the data type of the left-most integer. For example, all elements of the following vector are set to `int32`:

```
A = [true pi int32(1000000) single(17.32) uint8(250)]
```

Combining Cell Arrays with Non-Cell Arrays

Combining a number of arrays in which one or more is a cell array returns a new cell array. Each array being combined occupies a cell in the returned array:

```
A = [100, {uint8(200), 300}, 'MATLAB'];

whos A
  Name      Size              Bytes  Class     Attributes

  A         1x4                 269  cell
```

Each element of the combined array maintains its original class:

```
[t1 t2 t3 t4] = A{:};
nsprintf('   %s %s %s %s', class(t1), class(t2), ...
        class(t3), class(t4))
ans =
    double uint8 double char
```

Empty Matrices

If you construct a matrix using empty matrix elements, the empty matrices are ignored in the resulting matrix:

```
A = [5.36; 7.01; []; 9.44]
A =
    5.3600
    7.0100
    9.4400
```

Concatenation Examples

Here are some examples of data type conversion during matrix construction.

Combining Single and Double Types

Combining `single` values with `double` values yields a `single` matrix. Note that `5.73*10^300` is too big to be stored as a `single`, thus the conversion from `double` to `single` sets it to infinity. (The `class` function used in this example returns the data type for the input value).

```
x = [single(4.5) single(-2.8) pi 5.73*10^300]
x =
    4.5000   -2.8000    3.1416       Inf

class(x)              % Display the data type of x
ans =
    single
```

Combining Integer and Double Types

Combining integer values with `double` values yields an integer matrix. Note that the fractional part of `pi` is rounded to the nearest integer. (The `int8` function used in this example converts its numeric argument to an 8-bit integer).

```
x = [int8(21) int8(-22) int8(23) pi 45/6]
x =
    21   -22    23     3     7

class(x)
ans =
    int8
```

Combining Character and Double Types

Combining `character` values with `double` values yields a `character` matrix. MATLAB converts the `double` elements in this example to their `character` equivalents:

```
x = ['A' 'B' 'C' 68 69 70]
x =
   ABCDEF

class(x)
ans =
   char
```

Combining Logical and Double Types

Combining `logical` values with `double` values yields a `double` matrix. MATLAB converts the `logical` `true` and `false` elements in this example to `double`:

```
x = [true false false pi sqrt(7)]
x =
    1.0000         0         0    3.1416    2.6458

class(x)
ans =
   double
```

1-171

Defining Your Own Classes

All MATLAB data types are implemented as object-oriented classes. You can add data types of your own to your MATLAB environment by creating additional classes. These user-defined classes define the structure of your new data type, and the functions, or *methods*, that you write for each class define the behavior for that data type.

These methods can also define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the new data types. For example, a class called `polynomial` might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials.

With MATLAB classes you can

- Create methods that override existing MATLAB functionality
- Restrict the operations that are allowed on an object of a class
- Enforce common behavior among related classes by inheriting from the same parent class
- Significantly increase the reuse of your code

Read more about MATLAB classes in the MATLAB Classes and Object-Oriented Programming documentation.

2

Basic Program Components

MATLAB Commands

Basic Command Syntax

A simple MATLAB command computes the result of the expression to the right of the equals sign and assigns the value of the result to the output variable at the left. Examples of simple MATLAB commands are

```
x = 5.71;
A = [1 2 3; 4 5 6; 7 8 9];
I = besseli(nu, Z);
```

Commands that do not terminate with a semicolon display the result at your terminal as well as assigning it to the output variable:

```
A = [1 2 3; 4 5 6; 7 8 9]

A =
     1     2     3
     4     5     6
     7     8     9
```

If you do not explicitly assign the output of a command to a variable, MATLAB assigns the result to the reserved word `ans`:

```
[1 2 3; 4 5 6; 7 8 9]

ans =
     1     2     3
     4     5     6
     7     8     9
```

The value of `ans` changes with every command that returns an output value that is not assigned to a variable. We recommend that you do not use `ans` in place of variables as this practice tends to lead to code that is difficult to maintain and also to programming errors. The following use of `ans`, for example, is not recommended:

```
rand(3,5);
A = ans > 0.5;
```

Entering More Than One Command on a Line

You can enter more than one command on the same line, provided that you terminate each command with a comma or semicolon. Commands terminated with a comma display their results when they are executed; commands terminated with a semicolon do not:

```
rand('state', 0);
A = rand(3,5),    B = ones(3,5) * 4.7;   C = A./B

A =
    0.9501    0.4860    0.4565    0.4447    0.9218
    0.2311    0.8913    0.0185    0.6154    0.7382
    0.6068    0.7621    0.8214    0.7919    0.1763
C =
    0.2022    0.1034    0.0971    0.0946    0.1961
    0.0492    0.1896    0.0039    0.1309    0.1571
    0.1291    0.1621    0.1748    0.1685    0.0375
```

Assigning to Multiple Outputs

When a command generates more than one output, specify each output in square brackets to the left of the equals (=) sign. For example, the `deal` function distributes the values of each of its inputs to separate output variables:

```
[A, B, C] = deal([-12.3 4.89 -3.01], pi*1.46, diag(12:4:24))
A =
  -12.3000    4.8900   -3.0100
B =
    4.5867
C =
    12     0     0     0
```

```
     0    16     0     0
     0     0    20     0
     0     0     0    24
```

Other types of commands that can yield multiple outputs are assignments of structure and cell arrays, and calls to multiple-output functions. This example generates four outputs and assigns them to separate variables:

```
A(1).sym='H';  A(2).sym='He';  A(3).sym='Li';  A(4).sym='Be';
[hydrogen helium lithium beryllium] = A.sym

hydrogen =
    H
helium =
    He
lithium =
    Li
beryllium =
    Be
```

This example calls the `fileparts` function that returns several outputs, and assigns each output to a variable:

```
[path file exten] = ...
    fileparts('C:\matlab\work\strarray.mat')

path =
    C:\matlab\work
file =
    strarray
exten =
    .mat
```

For more information on assigning structure and cell arrays to multiple outputs, see "Assigning Struct Values to Separate Variables" on page 1-84 and "Assigning Cell Values to Separate Variables" on page 1-116.

For information on assigning functions to multiple outputs, see "Assigning Output Arguments" on page 3-53.

Assigning Fewer Than the Full Number of Outputs

When assigning structure and cell arrays or when calling multiple-output functions, if you specify fewer output variables than there are return values, MATLAB assigns one return value to each output variable specified and discards the rest. Repeat the last two examples shown above, but specify fewer than the full number of outputs that are available:

```
[Symbol_1 Symbol_2] = A.symb
Symbol_1 =
   H
Symbol_2 =
   He


[path file] = fileparts('..\work\strarray.mat')
path =
   C:\matlab\work
file =
   strarray
```

The `deal` function , however, does require that you specify the full number of output variables.

Commands that Call MATLAB Functions

When entering commands that call functions in MATLAB, you can use either of two syntaxes: *command* or *function* syntax. This is explained in the section "Command vs. Function Syntax" on page 3-31 in the MATLAB Programming Fundamentals documentation.

Expressions

String Evaluation

String evaluation adds power and flexibility to the MATLAB language, letting you perform operations like executing user-supplied strings and constructing executable strings through concatenation of strings stored in variables.

eval

The eval function evaluates a string that contains a MATLAB expression, statement, or function call. In its simplest form, the eval syntax is

```
eval('string')
```

For example, this code uses eval on an expression to generate a Hilbert matrix of order n.

```
t = '1/(m + n - 1)';
for m = 1:k
    for n = 1:k
        a(m,n) = eval(t);
    end
end
```

Here is an example that uses eval on a statement.

```
eval('t = clock');
```

Constructing Strings for Evaluation. You can concatenate strings to create a complete expression for input to eval. This code shows how eval can create 10 variables named P1, P2, ..., P10, and set each of them to a different value.

```
for n = 1:10
    eval(['P', int2str(n), '= n .^ 2'])
```

```
end
```

Shell Escape Functions

It is sometimes useful to access your own C or Fortran programs using *shell escape functions*. Shell escape functions use the shell escape command ! to make external stand-alone programs act like new MATLAB functions. A shell escape function

1 Saves the appropriate variables on disk.

2 Runs an external program (which reads the data file, processes the data, and writes the results back out to disk).

3 Loads the processed file back into the workspace.

For example, look at the code for `garfield.m`, below. This function uses an external function, `gareqn`, to find the solution to Garfield's equation.

```
function y = garfield(a,b,q,r)
save gardata a b q r
!gareqn
load gardata
```

This file

1 Saves the input arguments a, b, q, and r to a MAT-file in the workspace using the `save` command.

2 Uses the shell escape operator to access a C or Fortran program called `gareqn` that uses the workspace variables to perform its computation. `gareqn` writes its results to the `gardata` MAT-file.

3 Loads the `gardata` MAT-file described in "Custom Applications to Read and Write MAT-Files" to obtain the results.

Variables

Types of Variables

A MATLAB variable is essentially a tag that you assign to a value while that value remains in memory. The tag gives you a way to reference the value in memory so that your programs can read it, operate on it with other data, and save it back to memory.

MATLAB provides three basic types of variables:

- "Local Variables" on page 2-8
- "Global Variables" on page 2-9
- "Persistent Variables" on page 2-11

Local Variables

Each MATLAB function has its own local variables. These are separate from those of other functions (except for nested functions), and from those of the base workspace. Variables defined in a function do not remain in memory from one function call to the next, unless they are defined as `global` or `persistent`.

Scripts, on the other hand, do not have a separate workspace. They store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace.

Global Variables

If several functions, and possibly the base workspace, all declare a particular name as `global`, then they all share a single copy of that variable. Any assignment to that variable, in any function, is available to all the other functions declaring it global.

Suppose, for example, you want to study the effect of the interaction coefficients, α and β, in the Lotka-Volterra predator-prey model.

$$\dot{y}_1 = y_1 - \alpha y_1 y_2$$

$$\dot{y}_2 = -y_2 + \beta y_1 y_2$$

Create a program file, `lotka.m`.

```
function yp = lotka(t,y)
%LOTKA    Lotka-Volterra predator-prey model.
global ALPHA BETA
yp = [y(1) - ALPHA*y(1)*y(2); -y(2) + BETA*y(1)*y(2)];
```

Then interactively enter the statements

```
global ALPHA BETA
ALPHA = 0.01
BETA = 0.02
[t,y] = ode23(@lotka,[0,10],[1; 1]);
plot(t,y)
```

The two `global` statements make the values assigned to `ALPHA` and `BETA` at the command prompt available inside the function defined by `lotka.m`. They can be modified interactively and new solutions obtained without editing any files.

Creating Global Variables. Each function that uses a global variable must first declare the variable as `global`. It is usually best to put global declarations toward the beginning of the function. You would declare global variable MAXLEN as follows:

```
global MAXLEN
```

If the file contains subfunctions as well, then each subfunction requiring access to the global variable must declare it as `global`. To access the variable from the MATLAB command line, you must declare it as `global` at the command line.

MATLAB global variable names are typically longer and more descriptive than local variable names, and often consist of all uppercase characters. These are not requirements, but guidelines to increase the readability of MATLAB code, and to reduce the chance of accidentally redefining a global variable.

Displaying Global Variables. To see only those variables you have declared as global, use the `who` or `whos` functions with the literal, `global`.

```
global MAXLEN MAXWID
MAXLEN = 36;  MAXWID = 78;
len = 5;  wid = 21;

whos global
  Name          Size                   Bytes  Class

  MAXLEN        1x1                        8  double array (global)
  MAXWID        1x1                        8  double array (global)

Grand total is 2 elements using 16 bytes
```

Suggestions for Using Global Variables. A certain amount of risk is associated with using global variables and, because of this, it is recommended that you use them sparingly. You might, for example, unintentionally give a global variable in one function a name that is already used for a global variable in another function. When you run your application, one function may overwrite the variable used by the other. This error can be difficult to track down.

Another problem comes when you want to change the variable name. To make a change without introducing an error into the application, you must find every occurrence of that name in your code (and other people's code, if you share functions).

Alternatives to Using Global Variables. Instead of using a global variable, you may be able to

- Pass the variable to other functions as an additional argument. In this way, you make sure that any shared access to the variable is intentional.

   If this means that you have to pass a number of additional variables, you can put them into a structure or cell array and just pass it as one additional argument.

- Use a persistent variable (described in the next section), if you only need to make the variable persist in memory from one function call to the next.

Persistent Variables

Characteristics of persistent variables are

- You can declare and use them in functions only.

- Only the function in which the variables are declared is allowed access to it.

- MATLAB does not clear them from memory when the function exits, so their value is retained from one function call to the next.

You must declare persistent variables before you can use them in a function. It is usually best to put your persistent declarations toward the beginning of the function. You would declare persistent variable SUM_X as follows:

```
persistent SUM_X
```

If you clear a function that defines a persistent variable (i.e., using clear *functionname* or clear all), or if you edit the file for that function, MATLAB clears all persistent variables used in that function.

You can use the mlock function to keep a file from being cleared from memory, thus keeping persistent variables in the file from being cleared as well.

Initializing Persistent Variables. When you declare a persistent variable, MATLAB initializes its value to an empty matrix, []. After the declaration statement, you can assign your own value to it. This is often done using an `isempty` statement, as shown here:

```
function findSum(inputvalue)
persistent SUM_X

if isempty(SUM_X)
   SUM_X = 0;
end
SUM_X = SUM_X + inputvalue
```

This initializes the variable to 0 the first time you execute the function, and then it accumulates the value on each iteration.

Naming Variables

MATLAB variable names must begin with a letter, which may be followed by any combination of letters, digits, and underscores. MATLAB distinguishes between uppercase and lowercase characters, so `A` and `a` are not the same variable.

Although variable names can be of any length, MATLAB uses only the first `N` characters of the name, (where `N` is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first `N` characters to enable MATLAB to distinguish variables.

```
N = namelengthmax
N =
    63
```

The `genvarname` function can be useful in creating variable names that are both valid and unique. See the `genvarname` reference page to find out how to do this.

Verifying Validity of a Variable Name

Use the `isvarname` function to make sure a name is valid before you use it.
`isvarname` returns 1 if the name is valid, and 0 otherwise.

```
isvarname 8th_column
ans =
     0                     % Not valid - begins with a number
```

Verifying the Existence of a Variable

Use the `exist` function to see if a particular variable already exists in the
current workspace. `exist` returns 1 if a variable of that name is already in
the current workspace:

```
if ~exist('newvar', 'var')
    newvar = 10
else
    disp 'Variable already exists'
end

newvar =
    10
```

Avoid Using Function Names for Variables

When naming a variable, make sure you are not using a name that is already
used as a function name, either one of your own functions or one of the
functions in the MATLAB language. If you define a variable with a function
name, you will not be able to call that function until you either remove the
variable from memory with the `clear` function, or invoke the function using
`builtin`.

For example, if you enter the following command, you will not be able to use
the MATLAB `disp` function until you clear the variable with `clear disp`.

```
disp = 50;
```

To test whether a proposed variable name is already used as a function
name, use

```
which -all variable_name
```

Potential Conflict with Function Names

There are some MATLAB functions that have names that are commonly used as variable names in programming code. A few examples of such functions are i, j, mode, char, size, and path.

If you need to use a variable that is also the name of a MATLAB function, and have determined that you have no need to call the function, you should be aware that there is still a possibility for conflict. See the following two examples:

- "Variables Loaded From a MAT-File" on page 2-14
- "Variables In Evaluation Statements" on page 2-15

Variables Loaded From a MAT-File. The function shown below loads previously saved data from MAT-file settings.mat. It is supposed to display the value of one of the loaded variables, mode. However, mode is also the name of a MATLAB function and, in this case, MATLAB interprets it as the function and not the variable loaded from the MAT-file:

```
function show_mode
load settings;
whos mode
fprintf('Mode is set to %s\n', mode)
```

Assume that mode already exists in the MAT-file. Execution of the function shows that, even though mode is successfully loaded into the function workspace as a variable, when MATLAB attempts to operate on it in the last line, it interprets mode as a function. This results in an error:

```
show_mode
  Name        Size                      Bytes  Class

  mode        1x6                          12  char array

Grand total is 6 elements using 12 bytes

??? Error using ==> mode
```

```
Not enough input arguments.

Error in ==> show_mode at 4
fprintf('Mode is set to %s\n', mode)
```

Because MATLAB parses functions before they are run, it needs to determine before runtime which identifiers in the code are variables and which are functions. The function in this example does not establish mode as a variable name and, as a result, MATLAB interprets it as a function name instead.

There are several ways to make this function work as intended without having to change the variable name. Both indicate to MATLAB that the name represents a variable, and not a function:

- Name the variable explicitly in the load statement:

  ```
  function show_mode
  load settings mode;
  whos mode
  fprintf('Mode is set to %s\n', mode)
  ```

- Initialize the variable (e.g., set it to an empty matrix or empty string) at the start of the function:

  ```
  function show_mode
  mode = '';
  load settings;
  whos mode
  fprintf('Mode is set to %s\n', mode)
  ```

Variables In Evaluation Statements. Variables used in evaluation statements such as eval, evalc, and evalin can also be mistaken for function names. The following file defines a variable named length that conflicts with MATLAB length function:

```
function find_area
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

The second line of this code would seem to establish `length` as a variable name that would be valid when used in the statement on the third line. However, when MATLAB parses this line, it does not consider the *contents* of the string that is to be evaluated. As a result, MATLAB has no way of knowing that `length` was meant to be used as a variable name in this program, and the name defaults to a function name instead, yielding the following error:

```
find_area
??? Error using ==> length
Not enough input arguments.
```

To force MATLAB to interpret `length` as a variable name, use it in an explicit assignment statement first:

```
function find_area
length = [];
eval('length = 12;  width = 36;');
fprintf('The area is %d\n', length .* width)
```

Guidelines to Using Variables

The same guidelines that apply to MATLAB variables at the command line also apply to variables in your program files:

- You do not need to type or declare variables used in MATLAB program files (with the possible exception of designating them as `global` or `persistent`).

- Before assigning one variable to another, you must be sure that the variable on the right-hand side of the assignment has a value.

- Any operation that assigns a value to a variable creates the variable, if needed, or overwrites its current value, if it already exists.

Scope of a Variable

MATLAB stores variables in a part of memory called a workspace. The *base workspace* holds variables created during your interactive MATLAB session and also any variables created by running scripts. Variables created at the MATLAB command prompt can also be used by scripts without having to declare them as global.

Functions do not use the base workspace. Every function has its own *function workspace*. Each function workspace is kept separate from the base workspace and all other workspaces to protect the integrity of the data used by that function. Even subfunctions that are defined in the same file have a separate function workspace.

Extending Variable Scope

In most cases, variables created within a function are known only within that function. These variables are not available at the MATLAB command prompt or to any other function or subfunction.

Passing Variables from Another Workspace. The most secure way to extend the scope of a function variable is to pass it to other functions as an argument in the function call. Since MATLAB passes data only by value, you also need to add the variable to the return values of any function that modifies its value.

Evaluating in Another Workspace Using evalin. Functions can also obtain variables from either the base or the caller's workspace using the evalin function. The example below, compareAB_1, evaluates a command in the context of the MATLAB command line, taking the values of variables A and B from the base workspace.

Define A and B in the base workspace:

```
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Use evalin to evaluate the command A(find(A<=B)) in the context of the MATLAB base workspace:

```
function C = compareAB_1
C = evalin('base', 'A(find(A<=B))');
```

Call the function. You do not have to pass the variables because they are made available to the function via the evalin function:

```
C = compareAB_1
C =
    13     9    15
```

You can also evaluate in the context of the caller's workspace by specifying `'caller'` (instead of `'base'`) as the first input argument to `evalin`.

Using Global Variables. A third way to extend variable scope is to declare the variable as `global` within every function that needs access to it. If you do this, you need make sure that no functions with access to the variable overwrite its value unintentionally. For this reason, it is recommended that you limit the use of global variables.

Create global vectors `A` and `B` in the base workspace:

```
global A
global B
A = [13 25 82 68 9 15 77];  B = [63 21 71 42 30 15 22];
```

Also declare them in the function to be called:

```
function C = compareAB_2
global A
global B

C = A(find(A<=B));
```

Call the function. Again, you do not have to pass `A` and `B` as arguments to the called function:

```
C = compareAB_2
C =
    13     9    15
```

Scope in Nested Functions

Variables within nested functions are accessible to more than just their immediate function. As a general rule, the scope of a local variable is the largest containing function body in which the variable appears, and all functions nested within that function. For more information on nested functions, see "Variable Scope in Nested Functions" on page 4-19.

Lifetime of a Variable

Variables created at the MATLAB command prompt or in a script exist until you clear them or end your MATLAB session. Variables in functions exist only until the function completes unless they have been declared as `global` or `persistent`.

Keywords

The MATLAB software reserves certain words for its own use as keywords of the language. To list the keywords, type

```
iskeyword
ans =
    'break'
    'case'
    'catch'
        .
        .
        .
```

See the online function reference pages to learn how to use these keywords.

You should not use MATLAB keywords other than for their intended purpose. For example, a keyword should not be used as follows:

```
while = 5;
??? while = 5;
            |
Error: Expected a variable, function, or constant, found "=".
```

Special Values

Several functions return important special values that you can use in your own program files.

Function	Return Value
ans	Most recent answer (variable). If you do not assign an output variable to an expression, MATLAB automatically stores the result in ans.
eps	Floating-point relative accuracy. This is the tolerance the MATLAB software uses in its calculations.
intmax	Largest 8-, 16-, 32-, or 64-bit integer your computer can represent.
intmin	Smallest 8-, 16-, 32-, or 64-bit integer your computer can represent.
realmax	Largest floating-point number your computer can represent.
realmin	Smallest positive floating-point number your computer can represent.
pi	3.1415926535897...
i, j	Imaginary unit.
inf	Infinity. Calculations like n/0, where n is any nonzero real value, result in inf.
NaN	Not a Number, an invalid numeric value. Expressions like 0/0 and inf/inf result in a NaN, as do arithmetic operations involving a NaN. Also, if n is complex with a zero real part, then n/0 returns a value with a NaN real part.
computer	Computer type.
version	MATLAB version string.

Here are some examples that use these values in MATLAB expressions.

```
x = 2 * pi
x =
    6.2832

A = [3+2i 7-8i]
A =
   3.0000 + 2.0000i   7.0000 - 8.0000i

tol = 3 * eps
tol =
   6.6613e-016

intmax('uint64')
ans =
   18446744073709551615
```

Operators

Arithmetic Operators

Arithmetic operators perform numeric computations, for example, adding two numbers or raising the elements of an array to a given power. The following table provides a summary. For more information, see the arithmetic operators reference page.

Operator	Description
+	Addition
-	Subtraction
.*	Multiplication
./	Right division
.\	Left division
+	Unary plus
-	Unary minus
:	Colon operator
.^	Power
.'	Transpose
'	Complex conjugate transpose
*	Matrix multiplication
/	Matrix right division
\	Matrix left division
^	Matrix power

Arithmetic Operators and Arrays

Except for some matrix operators, MATLAB arithmetic operators work on corresponding elements of arrays with equal dimensions. For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. If one operand is a scalar and the other is not, MATLAB applies the scalar to every element of the other operand—this property is known as *scalar expansion*.

This example uses scalar expansion to compute the product of a scalar operand and a matrix.

```
A = magic(3)
A =
     8     1     6
     3     5     7
     4     9     2

3 * A
ans =
    24     3    18
     9    15    21
    12    27     6
```

Relational Operators

Relational operators compare operands quantitatively, using operators like "less than" and "not equal to." The following table provides a summary. For more information, see the relational operators reference page.

Operator	Description
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
~=	Not equal to

Relational Operators and Arrays

The MATLAB relational operators compare corresponding elements of arrays with equal dimensions. Relational operators always operate element-by-element. In this example, the resulting matrix shows where an element of A is equal to the corresponding element of B.

```
A = [2 7 6;9 0 5;3 0.5 6];
B = [8 7 0;3 2 5;4 -1 7];

A == B
ans =
     0     1     0
     0     0     1
     0     0     0
```

For vectors and rectangular arrays, both operands must be the same size unless one is a scalar. For the case where one operand is a scalar and the other is not, MATLAB tests the scalar against every element of the other operand. Locations where the specified relation is true receive logical 1. Locations where the relation is false receive logical 0.

Relational Operators and Empty Arrays

The relational operators work with arrays for which any dimension has size zero, as long as both arrays are the same size or one is a scalar. However, expressions such as

```
A == []
```

return an error if A is not 0-by-0 or 1-by-1. This behavior is consistent with that of all other binary operators, such as +, -, >, <, &, |, etc.

To test for empty arrays, use the function

```
isempty(A)
```

Logical Operators

MATLAB offers three types of logical operators and functions:

• Element-wise — operate on corresponding elements of logical arrays.

- Bit-wise — operate on corresponding bits of integer values or arrays.

- Short-circuit — operate on scalar, logical expressions.

The values returned by MATLAB logical operators and functions, with the exception of bit-wise functions, are of type `logical` and are suitable for use with logical indexing.

Element-Wise Operators and Functions

The following logical operators and functions perform elementwise logical operations on their inputs to produce a like-sized output array.

The examples shown in the following table use vector inputs A and B, where

```
A = [0 1 1 0 1];
B = [1 1 0 0 1];
```

Operator	Description	Example
&	Returns 1 for every element location that is true (nonzero) in both arrays, and 0 for all other elements.	A & B = 01001
|	Returns 1 for every element location that is true (nonzero) in either one or the other, or both arrays, and 0 for all other elements.	A | B = 11101
~	Complements each element of the input array, A.	~A = 10010
xor	Returns 1 for every element location that is true (nonzero) in only one array, and 0 for all other elements.	xor(A,B) = 10100

For operators and functions that take two array operands, (&, |, and xor), both arrays must have equal dimensions, with each dimension being the same size. The one exception to this is where one operand is a scalar and the other is not. In this case, MATLAB tests the scalar against every element of the other operand.

Note MATLAB converts any finite nonzero, numeric values used as inputs to logical expressions to logical 1, or true.

Operator Overloading. You can overload the &, |, and ~ operators to make their behavior dependent upon the class on which they are being used. Each of these operators has a representative function that is called whenever that operator is used. These are shown in the table below.

Logical Operation	Equivalent Function
A & B	and(A, B)
A | B	or(A, B)
~A	not(A)

Other Array Functions. Two other MATLAB functions that operate logically on arrays, but not in an elementwise fashion, are any and all. These functions show whether *any* or *all* elements of a vector, or a vector within a matrix or an array, are nonzero.

When used on a matrix, any and all operate on the columns of the matrix. When used on an N-dimensional array, they operate on the first nonsingleton dimension of the array. Or, you can specify an additional dimension input to operate on a specific dimension of the array.

The examples shown in the following table use array input A, where

```
A = [0    1    2;
     0   -3    8;
     0    5    0];
```

Function	Description	Example
any(A)	Returns 1 for a vector where *any* element of the vector is true (nonzero), and 0 if no elements are true.	any(A) ans = 0 1 1
all(A)	Returns 1 for a vector where *all* elements of the vector are true (nonzero), and 0 if all elements are not true.	all(A) ans = 0 1 0

Note The all and any functions ignore any NaN values in the input arrays.

Short-Circuiting in Elementwise Operators. When used in the context of an if or while expression, and only in this context, the elementwise | and & operators use short-circuiting in evaluating their expressions. That is, A|B and A&B ignore the second operand, B, if the first operand, A, is sufficient to determine the result.

So, although the statement 1|[] evaluates to false, the same statement evaluates to true when used in either an if or while expression:

```
A = 1;   B = [];
if(A|B) disp 'The statement is true',  end;
    The statement is true
```

while the reverse logical expression, which does not short-circuit, evaluates to false

```
if(B|A) disp 'The statement is true',  end;
```

Another example of short-circuiting with elementwise operators shows that a logical expression such as the following, which under most circumstances is invalid due to a size mismatch between A and B,

```
A = [1 1];   B = [2 0 1];
A|B        % This generates an error.
```

works within the context of an if or while expression:

```
if (A|B) disp 'The statement is true',  end;
```

```
        The statement is true
```

Logical Expressions Using the find Function. The `find` function
determines the indices of array elements that meet a given logical condition.
The function is useful for creating masks and index matrices. In its most
general form, `find` returns a single vector of indices. This vector can be used
to index into arrays of any size or shape.

For example,

```
A = magic(4)
A =
    16     2     3    13
     5    11    10     8
     9     7     6    12
     4    14    15     1

i = find(A > 8);
A(i) = 100
A =
   100     2     3   100
     5   100   100     8
   100     7     6   100
     4   100   100     1
```

Note An alternative to using `find` in this context is to index into the matrix
using the logical expression itself. See the example below.

The last two statements of the previous example can be replaced with this
one statement:

```
A(A > 8) = 100;
```

You can also use `find` to obtain both the row and column indices of a
rectangular matrix for the array values that meet the logical condition:

```
A = magic(4)
A =
    16     2     3    13
```

```
     5    11    10     8
     9     7     6    12
     4    14    15     1

[row, col] = find(A > 12)
row =
     1
     4
     4
     1
col =
     1
     2
     3
     4
```

Bit-Wise Functions

The following functions perform bit-wise logical operations on nonnegative integer inputs. Inputs may be scalar or in arrays. If in arrays, these functions produce a like-sized output array.

The examples shown in the following table use scalar inputs A and B, where

```
A = 28;                  % binary 11100
B = 21;                  % binary 10101
```

Function	Description	Example
bitand	Returns the bit-wise AND of two nonnegative integer arguments.	bitand(A,B) = 20 (binary 10100)
bitor	Returns the bit-wise OR of two nonnegative integer arguments.	bitor(A,B) = 29 (binary 11101)

Function	Description	Example
bitcmp	Returns the bit-wise complement as an n-bit number, where n is the second input argument to bitcmp.	bitcmp(A,5) = 3 (binary 00011)
bitxor	Returns the bit-wise exclusive OR of two nonnegative integer arguments.	bitxor(A,B) = 9 (binary 01001)

Short-Circuit Operators

The following operators perform AND and OR operations on logical expressions containing scalar values. They are *short-circuit* operators in that they evaluate their second operand only when the result is not fully determined by the first operand.

Operator	Description
&&	Returns logical 1 (true) if both inputs evaluate to true, and logical 0 (false) if they do not.
||	Returns logical 1 (true) if either input, or both, evaluate to true, and logical 0 (false) if they do not.

The statement shown here performs an AND of two logical terms, A and B:

```
A && B
```

If A equals zero, then the entire expression will evaluate to logical 0 (false), regardless of the value of B. Under these circumstances, there is no need to evaluate B because the result is already known. In this case, MATLAB short-circuits the statement by evaluating only the first term.

A similar case is when you OR two terms and the first term is true. Again, regardless of the value of B, the statement will evaluate to true. There is no need to evaluate the second term, and MATLAB does not do so.

Advantage of Short-Circuiting. You can use the short-circuit operators to evaluate an expression only when certain conditions are satisfied. For example, you want to execute a function only if the function file resides on the current MATLAB path.

Short-circuiting keeps the following code from generating an error when the file, `myfun.m`, cannot be found:

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

Similarly, this statement avoids attempting to divide by zero:

```
x = (b ~= 0) && (a/b > 18.5)
```

You can also use the `&&` and `||` operators in `if` and `while` statements to take advantage of their short-circuiting behavior:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

Operator Precedence

You can build expressions that use any combination of arithmetic, relational, and logical operators. Precedence levels determine the order in which MATLAB evaluates an expression. Within each precedence level, operators have equal precedence and are evaluated from left to right. The precedence rules for MATLAB operators are shown in this list, ordered from highest precedence level to lowest precedence level:

1 Parentheses `()`

2 Transpose `(.')`, power `(.^)`, complex conjugate transpose `(')`, matrix power `(^)`

3 Unary plus `(+)`, unary minus `(-)`, logical negation `(~)`

4 Multiplication `(.*)`, right division `(./)`, left division `(.\)`, matrix multiplication `(*)`, matrix right division `(/)`, matrix left division `(\)`

5 Addition `(+)`, subtraction `(-)`

6 Colon operator `(:)`

7 Less than (<), less than or equal to (<=), greater than (>), greater than or equal to (>=), equal to (==), not equal to (~=)

8 Element-wise AND (&)

9 Element-wise OR (|)

10 Short-circuit AND (&&)

11 Short-circuit OR (||)

Precedence of AND and OR Operators

MATLAB always gives the & operator precedence over the | operator. Although MATLAB typically evaluates expressions from left to right, the expression a|b&c is evaluated as a|(b&c). It is a good idea to use parentheses to explicitly specify the intended precedence of statements containing combinations of & and |.

The same precedence rule holds true for the && and || operators.

Overriding Default Precedence

The default precedence can be overridden using parentheses, as shown in this example:

```
A = [3 9 5];
B = [2 1 5];
C = A./B.^2
C =
    0.7500    9.0000    0.2000

C = (A./B).^2
C =
    2.2500   81.0000    1.0000
```

Comma-Separated Lists

What Is a Comma-Separated List?

Typing in a series of numbers separated by commas gives you what is called a *comma-separated list*. The MATLAB software returns each value individually:

```
1, 2, 3
ans =
      1
ans =
      2
ans =
      3
```

Such a list, by itself, is not very useful. But when used with large and more complex data structures like MATLAB structures and cell arrays, the comma-separated list can enable you to simplify your MATLAB code.

Generating a Comma-Separated List

This section describes how to generate a comma-separated list from either a cell array or a MATLAB structure.

Generating a List from a Cell Array

Extracting multiple elements from a cell array yields a comma-separated list. Given a 4-by-6 cell array as shown here

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;    end

C
C =
    [2]     [10]     [18]     [26]     [34]     [42]
    [4]     [12]     [20]     [28]     [36]     [44]
    [6]     [14]     [22]     [30]     [38]     [46]
    [8]     [16]     [24]     [32]     [40]     [48]
```

extracting the fifth column generates the following comma-separated list:

```
C{:, 5}
ans =
    34
ans =
    36
ans =
    38
ans =
    40
```

This is the same as explicitly typing

```
C{1, 5}, C{2, 5}, C{3, 5}, C{4, 5}
```

Generating a List from a Structure

For structures, extracting a field of the structure that exists across one of its dimensions yields a comma-separated list.

Start by converting the cell array used above into a 4-by-1 MATLAB structure with six fields: f1 through f6. Read field f5 for all rows and MATLAB returns a comma-separated list:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

S.f5
ans =
    34
ans =
```

```
             36
      ans =
             38
      ans =
             40
```

This is the same as explicitly typing

```
S(1).f5, S(2).f5, S(3).f5, S(4).f5
```

Assigning Output from a Comma-Separated List

You can assign any or all consecutive elements of a comma-separated list to
variables with a simple assignment statement. Using the cell array C from
the previous section, assign the first row to variables c1 through c6:

```
C = cell(4, 6);
for k = 1:24,   C{k} = k * 2;   end

[c1 c2 c3 c4 c5 c6] = C{1,1:6};

c5
c5 =
    34
```

If you specify fewer output variables than the number of outputs returned by
the expression, MATLAB assigns the first N outputs to those N variables, and
then discards any remaining outputs. In this next example, MATLAB assigns
C{1,1:3} to the variables c1, c2, and c3, and then discards C{1,4:6}:

```
[c1 c2 c3] = C{1,1:6};
```

You can assign structure outputs in the same manner:

```
S = cell2struct(C, {'f1', 'f2', 'f3', 'f4', 'f5', 'f6'}, 2);

[sf1 sf2 sf3] = S.f5;

sf3
sf3 =
    38
```

You also can use the `deal` function for this purpose.

Assigning to a Comma-Separated List

The simplest way to assign multiple values to a comma-separated list is to use the `deal` function. This function distributes all of its input arguments to the elements of a comma-separated list.

This example initializes a comma-separated list to a set of vectors in a cell array, and then uses `deal` to overwrite each element in the list:

```
c{1} = [31 07];    c{2} = [03 78];

c{:}
ans =
    31     7
ans =
     3    78


[c{:}] = deal([10 20],[14 12]);

c{:}
ans =
    10    20
ans =
    14    12
```

This example does the same as the one above, but with a comma-separated list of vectors in a structure field:

```
s(1).field1 = [31 07];    s(2).field1 = [03 78];

s.field1
ans =
    31     7
ans =
     3    78
```

```
[s.field1] = deal([10 20],[14 12]);

s.field1
ans =
    10    20
ans =
    14    12
```

How to Use the Comma-Separated Lists

Common uses for comma-separated lists are

- "Constructing Arrays" on page 2-38
- "Displaying Arrays" on page 2-39
- "Concatenation" on page 2-39
- "Function Call Arguments" on page 2-39
- "Function Return Values" on page 2-40

The following sections provide examples of using comma-separated lists with cell arrays. Each of these examples applies to MATLAB structures as well.

Constructing Arrays

You can use a comma-separated list to enter a series of elements when constructing a matrix or array. Note what happens when you insert a *list* of elements as opposed to adding the cell itself.

When you specify a list of elements with C{:, 5}, MATLAB inserts the four individual elements:

```
A = {'Hello', C{:, 5}, magic(4)}
A =
    'Hello'    [34]    [36]    [38]    [40]    [4x4 double]
```

When you specify the C cell itself, MATLAB inserts the entire cell array:

```
A = {'Hello', C, magic(4)}
A =
    'Hello'    {4x6 cell}    [4x4 double]
```

Displaying Arrays

Use a list to display all or part of a structure or cell array:

```
A{:}
ans =
    Hello
ans =
    34
ans =
    36
ans =
    38
  .
  .
  .
```

Concatenation

Putting a comma-separated list inside square brackets extracts the specified elements from the list and concatenates them:

```
A = [C{:, 5:6}]
A =
    34    36    38    40    42    44    46    48

whos A
  Name      Size                      Bytes  Class

  A         1x8                          64  double array
```

Function Call Arguments

When writing the code for a function call, you enter the input arguments as a list with each argument separated by a comma. If you have these arguments stored in a structure or cell array, then you can generate all or part of the argument list from the structure or cell array instead. This can be especially useful when passing in variable numbers of arguments.

This example passes several attribute-value arguments to the `plot` function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';        C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';  C{2,2} = 'k';
C{1,3} = 'MarkerFaceColor';  C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Function Return Values

MATLAB functions can also return more than one value to the caller. These values are returned in a list with each value separated by a comma. Instead of listing each return value, you can use a comma-separated list with a structure or cell array. This becomes more useful for those functions that have variable numbers of return values.

This example returns three values to a cell array:

```
C = cell(1, 3);
[C{:}] = fileparts('work/mytests/strArrays.mat')
C =
    'work/mytests'    'strArrays'    '.mat'
```

Fast Fourier Transform Example

The `fftshift` function swaps the left and right halves of each dimension of an array. For a simple vector such as [0 2 4 6 8 10] the output would be [6 8 10 0 2 4]. For a multidimensional array, `fftshift` performs this swap along each dimension.

`fftshift` uses vectors of indices to perform the swap. For the vector shown above, the index [1 2 3 4 5 6] is rearranged to form a new index [4 5 6 1 2 3]. The function then uses this index vector to reposition the elements. For a multidimensional array, `fftshift` must construct an index vector for each dimension. A comma-separated list makes this task much simpler.

Here is the `fftshift` function:

```
function y = fftshift(x)
```

```
numDims = ndims(x);
idx = cell(1, numDims);

for k = 1:numDims
   m = size(x, k);
   p = ceil(m/2);
   idx{k} = [p+1:m 1:p];
   end

y = x(idx{:});
```

The function stores the index vectors in cell array `idx`. Building this cell array is relatively simple. For each of the `N` dimensions, determine the size of that dimension and find the integer index nearest the midpoint. Then, construct a vector that swaps the two halves of that dimension.

By using a cell array to store the index vectors and a comma-separated list for the indexing operation, `fftshift` shifts arrays of any dimension using just a single operation: `y = x(idx{:})`. If you were to use explicit indexing, you would need to write one `if` statement for each dimension you want the function to handle:

```
if ndims(x) == 1
   y = x(index1);
else if ndims(x) == 2
   y = x(index1, index2);
end
```

Another way to handle this without a comma-separated list would be to loop over each dimension, converting one dimension at a time and moving data each time. With a comma-separated list, you move the data just once. A comma-separated list makes it very easy to generalize the swapping operation to an arbitrary number of dimensions.

Program Control Statements

Conditional Control — if, switch

This group of control statements enables you to select at run-time which block of code is executed. To make this selection based on whether a condition is true or false, use the if statement (which may include else or elseif). To select from a number of possible options depending on the value of an expression, use the switch and case statements (which may include otherwise).

You cannot define nested functions within a conditional control block. Nested functions must always be defined at the top level of a function.

if, else, and elseif

if evaluates a logical expression and executes a group of statements based on the value of the expression. In its simplest form, its syntax is

```
if logical_expression
    statements
end
```

If the logical expression is true (that is, if it evaluates to logical 1), the MATLAB software executes all the statements between the if and end lines. It resumes execution at the line following the end statement. If the condition is false (evaluates to logical 0), MATLAB skips all the statements between the if and end lines, and resumes execution at the line following the end statement.

For example,

```
if rem(a, 2) == 0
    disp('a is even')
    b = a/2;
end
```

You can nest any number of `if` statements.

If the logical expression evaluates to a nonscalar value, all the elements of the argument must be nonzero. For example, assume X is a matrix. Then the statement

```
if X
    statements
end
```

is equivalent to

```
if all(X(:))
    statements
end
```

The `else` and `elseif` statements further conditionalize the `if` statement:

- The `else` statement has no logical condition. The statements associated with it execute if the preceding `if` (and possibly `elseif` condition) evaluates to logical 0 (`false`).

- The `elseif` statement has a logical condition that it evaluates if the preceding `if` (and possibly `elseif` condition) is `false`. The statements associated with it execute if its logical condition evaluates to logical 1 (`true`). You can have multiple `elseif` statements within an `if` block.

```
if n < 0            % If n negative, display error message.
    disp('Input must be positive');
elseif rem(n,2) == 0 % If n positive and even, divide by 2.
    A = n/2;
else
    A = (n+1)/2;    % If n positive and odd, increment and divide.
end
```

if Statements and Empty Arrays. An `if` condition that reduces to an empty array represents a `false` condition. That is,

```
if A
    S1
else
    S0
end
```

executes statement S0 when A is an empty array.

switch, case, and otherwise

switch executes certain statements based on the value of a variable or expression. Its basic form is

```
switch expression (scalar or string)
    case value1
        statements        % Executes if expression is value1
    case value2
        statements        % Executes if expression is value2
      .
       .
        .
    otherwise
        statements        % Executes if expression does not
                           % match any case
end
```

This block consists of

- The word switch followed by an expression to evaluate.

- Any number of case groups. These groups consist of the word case followed by a possible value for the expression, all on a single line. Subsequent lines contain the statements to execute for the given value of the expression. These can be any valid MATLAB statement including another switch block. Execution of a case group ends when MATLAB encounters the next case statement or the otherwise statement. Only the first matching case is executed.

- An optional otherwise group. This consists of the word otherwise, followed by the statements to execute if the expression's value is not handled by any of the preceding case groups. Execution of the otherwise group ends at the end statement.

- An `end` statement.

`switch` works by comparing the input expression to each case value. For numeric expressions, a case statement is true if (`value==expression`). For string expressions, a case statement is true if `strcmp(value,expression)`.

The code below shows a simple example of the `switch` statement. It checks the variable input_num for certain values. If input_num is -1, 0, or 1, the `case` statements display the value as text. If input_num is none of these values, execution drops to the `otherwise` statement and the code displays the text `'other value'`.

```
switch input_num
    case -1
        disp('negative one');
    case 0
        disp('zero');
    case 1
        disp('positive one');
    otherwise
         disp('other value');
end
```

Note For C programmers, unlike the C language `switch` construct, the MATLAB `switch` does not "fall through." That is, if the first `case` statement is true, other `case` statements do not execute. Therefore, `break` statements are not used.

`switch` can handle multiple conditions in a single `case` statement by enclosing the case expression in a cell array.

```
switch var
    case 1
        disp('1')
    case {2,3,4}
        disp('2 or 3 or 4')
    case 5
        disp('5')
    otherwise
```

```
            disp('something else')
    end
```

Loop Control — for, while, continue, break

With loop control statements, you can repeatedly execute a block of code, looping back through the block while keeping track of each iteration with an incrementing index variable. Use the for statement to loop a specific number of times. The while statement is more suitable for basing the loop execution on how long a condition continues to be true or false. The continue and break statements give you more control on exiting the loop.

You cannot define nested functions within a loop control block. Nested functions must always be defined at the top level of a function.

Note You can often speed up the execution of MATLAB code by replacing for and while loops with vectorized code. See "Techniques for Improving Performance" on page 8-4 for more information on this.

---

for

The for loop executes a statement or group of statements a predetermined number of times. Its syntax is

```
for index = start:increment:end
    statements
end
```

The default increment is 1. You can specify any increment, including a negative one. For positive indices, execution terminates when the value of the index exceeds the *end* value; for negative increments, it terminates when the index is less than the end value.

For example, this loop executes five times.

```
for n = 2:6
    x(n) = 2 * x(n - 1);
end
```

You can nest multiple `for` loops.

```
for m = 1:5
    for n = 1:100
        A(m, n) = 1/(m + n - 1);
    end
end
```

Note You can often speed up the execution of MATLAB code by replacing `for` and `while` loops with vectorized code. See "Vectorizing Loops" on page 8-8 for details.

Using Arrays as Indices. The index of a `for` loop can be an array. For example, consider an m-by-n array A. The statement

```
for k = A
    statements
end
```

sets k equal to the vector `A(:,i)`, where i is the iteration number of the loop. For the first loop iteration, k is equal to `A(:,1)`; for the second, k is equal to `A(:,2)`; and so on until k equals `A(:,n)`. That is, the loop iterates for a number of times equal to the number of columns in A. For each iteration, k is a vector containing one of the columns of A.

while

The `while` loop executes a statement or group of statements repeatedly as long as the controlling expression is true (1). Its syntax is

```
while expression
    statements
end
```

If the expression evaluates to a matrix, all its elements must be 1 for execution to continue. To reduce a matrix to a scalar value, use the `all` and `any` functions.

For example, this `while` loop finds the first integer n for which n! (n factorial) is a 100-digit number.

```
n = 1;
while prod(1:n) < 1e100
    n = n + 1;
end
```

Exit a `while` loop at any time using the `break` statement.

while Statements and Empty Arrays. A `while` condition that reduces to an empty array represents a `false` condition. That is,

```
while A, S1, end
```

never executes statement `S1` when `A` is an empty array.

continue

The `continue` statement passes control to the next iteration of the `for` or `while` loop in which it appears, skipping any remaining statements in the body of the loop. In `for` loops, the loop counter is incremented by the appropriate value (either 1 or the specified step value) at the start of the next iteration.

`continue` works the same way in nested loops. That is, execution continues at the beginning of the loop in which the `continue` statement was encountered.

The example below shows a `continue` loop that counts the lines of code in the file, `magic.m`, skipping all blank lines and comments. A `continue` statement is used to advance to the next line in `magic.m` without incrementing the count whenever a blank line or comment line is encountered.

```
fid = fopen('magic.m','r');
count = 0;
while ~feof(fid)
    line = fgetl(fid);
    if isempty(line) || strncmp(line,'%',1) || ~ischar(line)
        continue
    end
    count = count + 1;
end
```

```
   fprintf('%d lines\n',count);
   fclose(fid);
```

break

The `break` statement terminates the execution of a `for` loop or `while` loop. When a `break` statement is encountered, execution continues with the next statement outside of the loop. In nested loops, `break` exits from the innermost loop only.

The example below shows a `while` loop that reads the contents of the file `fft.m` into a MATLAB character array. A `break` statement is used to exit the `while` loop when the first empty line is encountered. The resulting character array contains the help for the `fft` program.

```
fid = fopen('fft.m','r');
s = '';

while ~feof(fid)
   line = fgetl(fid);
   if isempty(line) || ~ischar(line)
      break
   end
   s = sprintf('%s%s\n', s, line);
end
disp(s);

fclose(fid);
```

Error Control — try, catch

Error control statements provide a way for you to take certain actions in the event of an error. Use the `try` statement to test whether certain commands in your code generates an error. If an error does occur within the `try` block, MATLAB immediately jumps to the corresponding `catch` block. Use the `catch` part of the statement to respond in some way to the error.

You cannot define nested functions within a `try-catch` block. Nested functions must always be defined at the top level of a function.

try and catch

The general form of a try-catch statement sequence is

```
try
    statement
    ...
    statement
catch meObj
    statement
    ...
    statement
end
```

In this sequence, the statements in the `try` block (that part of the try-catch that follows the word `try` statement, and precedes `catch`) between `try` and `catch` execute just like any other program code. If an error occurs within the `try` section The statements between `catch` and `end` are then executed. Examine the contents of the `MException` object `meObj` to see the cause of the error. If an error occurs between `catch` and `end`, MATLAB terminates execution unless another `try-catch` sequence has been established.

Program Termination — return

Program termination control enables you to exit from your program at some point prior to its normal termination point.

return

After a MATLAB function runs to completion, it terminates and returns control either to the function that called it, or to the keyboard. If you need to exit a function prior to the point of normal completion, you can force an early termination using the `return` function. `return` immediately terminates the current sequence of commands and exits the currently running function.

`return` is also used to terminate `keyboard` mode.

Dates and Times

Representing Dates and Times in MATLAB

The MATLAB software represents date and time information in any of three formats:

- Date String — A character string for which you select which fields to return, and how you want these fields to appear in the output string.

    ```
    Example:  Wednesday, August 23, 2010  10:35:42.946 AM
    ```

- Date Vector — A 1-by-6 numeric vector containing the year, month, day, hour, minute, and second.

    ```
    Example:  [2010   5    25    9    45    44.9]
    ```

- Serial Date Number — A single number equal to the number of days since a fixed, preset date (January 0, 0000).

    ```
    Example:  7.3428e+005
    ```

You have the choice of using any of these formats. If you work with more than one date and time format, MATLAB provides functions to help you easily convert from one format to another.

Date and Time Functions

This table shows what information is available in MATLAB with respect to dates and times and which function provides this information. The sections that follow the table provide more information on how to use the different date and time formats:

Date and Time Information	Output Format	Function to Use
Current date and time	Date number	`now`
	Date vector	`clock`
	Date string	`datestr(now)`
Current date	Date number	`datenum(date)`
	Date vector	`datevec(date)`
	Date string	`date`
Day of week for given date	Full day name, abbreviated name, or day number in week (1-7)	`weekday`
Last day of given month(s)	Vector of one or more days	`eomday`
Date with modified field	Date number	`addtodate`
Calendar for given month	6-by-7 matrix of days	`calendar`

For examples showing how to use these functions, see the function reference documentation.

Working with Date Strings

There are a number of ways to represent dates and times in character string format.For example, all of the following are date strings for August 23, 2010 at 04:35:42 PM.:

```
'23-Aug-2010 04:35:06 PM'
'Wednesday, August 23'
'08/23/10 16:35'
```

```
'Aug 23 16:35:42.946'
```

Creating Date Strings In MATLAB

A date string is a character string composed of fields related to a specific date and/or time. In its default form, a date string has six fields containing values for a specific day, month, year, hour, minute, and second, in that order:

```
'23-Aug-2010 16:35:10'
```

A default date string that contains just a date consists of three fields: day, month, and year. The `date` function returns this type of date string:

```
date
ans =
    23-Aug-2010
```

To create a date string, you simply enter it as a MATLAB character string. Include any characters you might need to separate the fields, such as the hyphen, space, and colon used here:

```
d = '23-Aug-2010 16:35:42'
```

Minimum Requirements for Date Strings. A date string must contain at least a month and day field, or an hour and minute field. When entering month and day fields, MATLAB expects the month to precede the day. (There are ways to reverse this order, but that is more advanced usage and is documented under "How to Use the Field Specifier" on page 2-54.) The following date string is August 12, not December 8:

```
d = '08/12'
```

When entering the hour and minute fields, put the hour first and separate the two fields with a colon. The following date strings are 10:35 in the morning and 3:20 in the afternoon in both 24-hour and 12-hour notation:

```
t = '10:35'      t = '10:35 AM'
t = '15:20'      t = '3:20 PM'
```

If the date includes the year, then the year value must immediately precede or immediately follow the month and day. The following are both valid date strings that represent the same day, August 12 in the year 2010:

```
d = '08/12/2010'
d = '2010/08/12'
```

When unspecified, the year defaults to the current year, month and day default to January 1, hour and minute default to 00:00, and second and millisecond default to 00.000.

Specifying the Fields of a Date String

The MATLAB software provides the `datevec`, `datenum`, and `datestr` functions for converting from one date format to another. When you pass a date string to one of these functions, MATLAB does not necessarily know which fields have been included in the string, or the order in which they are positioned. Also, if you expect a date string to be returned by one of the conversion functions, you might need to indicate how that string is to be composed. For these reasons, unless you are passing a date string that uses the default field format, MathWorks recommends that you pass an additional argument called a *field specifier* in the call.

How to Use the Field Specifier. The reference page syntax for date conversion functions identifies the field specifier arguments as *FieldSpecIn* and *FieldSpecOut*. The `datevec` and `datenum` functions use the *FieldSpecIn* argument to indicate how MATLAB is to interpret the *input* date string:

```
DateVector = datevec(DateString, FieldSpecIn)
```

The `datestr` function uses the *FieldSpecOut* argument to indicate how MATLAB is to display the *output* date string:

```
DateString = datestr(DateVector, FieldSpecOut)
```

If you need to use a field specifier for both input and output strings, nest a call to `datenum` inside a call to `datestr` or `datevec`. This example changes date string `'August 11'` to `'11 August'`. Because neither string is in the default format, you need a field specifier for both the input and output date strings:

```
datestr(datenum('August 11', 'mmmm dd'), 'dd mmmm')
ans =
```

```
11 August
```

Note To convert a nonstandard date form into a MATLAB date form, first convert the nonstandard date form to a date number.

Character-Based Field Identifiers. There are two types of identifiers with which you can indicate the layout of fields in a date string. The more general of the two employs character-based symbols, such as those shown in this table, to designate the fields of a date string. (See "Symbolic Identifiers for Individual Fields" on page 2-58 for the full table):

Field	Example	Identifier
Year (in 4 digits)	2010	yyyy
Month (in 3 characters)	Aug	mmm
Day Number	23	dd
Hour	16	HH
Minute	35	MM
Second	42	SS

The goal of the following example is to convert the date vector [2010 08 23 16 35 00] to a date string that has the nondefault format:

```
2010-08-23 16:35:42
```

Look up each of the six date fields in the table and use the symbols you find to compose a field specifier string that tells MATLAB how you want the date string output to look:

```
InputVector = [2010 08 23 16 35 42];

datestr(InputVector, 'yyyy-mm-dd HH:MM:SS')
ans =
    2010-08-23 16:35:42
```

Numeric Field Identifiers. There is an additional method for indicating the type of format you want applied to a date string. The MATLAB software associates 31 commonly used field sequences with a numeric identifier for each. Using these predefined formats can simplify the creation of your date strings. The table below shows several of these identifiers and the type of date string associated with them. (See "Numeric Identifiers for Predefined Formats" on page 2-60 for the full table):

Date String Formats	Example of Output	Identifier to Use
`'dd-mmm-yyyy'`	`01-Mar-2000`	1
`'HH:MM:SS PM'`	`3:45:17 PM`	14
`'mm/dd/yyyy'`	`03/01/2000`	23
`'dd/mm/yyyy'`	`01/03/2000`	24
`'yyyy-mm-dd HH:MM:SS'`	`2000-03-01 15:45:17`	31

Repeat the example from the previous section, this time using a numeric field specifier:

```
InputVector = [2010 08 23 16 35 42];

datestr(InputVector, 31)
ans =
    2010-08-23 16:35:42
```

Compare the commands used to achieve the same result. The advantage of the character-based field specifiers is that they are more versatile and easier to remember. The advantage of the numeric field specifiers is that frequently used commands are easier to enter:

```
datestr(InputVector, 'yyyy-mm-dd HH:MM:SS')    % Character
datestr(InputVector, 31)                       % Numeric
```

Adding Field Separation Characters. It is customary to separate certain fields of a date string with some form of punctuation, such as commas or space characters. Insert these characters into a date string by including them in the field specifier string. You cannot use any characters that could conflict with those symbolic characters reserved for use in the `FieldSpecIn` or `fieldSpecOut` strings (for example, y, m, H, M, and so on).

Add separation characters to make a date string easier to read:

```
datestr(now, 'dddd, mmmm dd, yyyy HH:MM')
ans =
    Wednesday, August 23, 2010 16:35
```

Creating Multiple Date Strings

Calling `datestr` with more than one date string input returns a character array of converted date strings. Pass the multiple date strings in a cell array. All input date strings must use the same format. For example, the following command passes three dates that all use the mm/dd/yyyy format:

```
datestr(datenum({'09/16/2007';'05/14/1996';'11/29/2010'}, ...
    'mm/dd/yyyy'))
ans =
    16-Sep-2007
    14-May-1996
    29-Nov-2010
```

Time Display in Date Strings

In MATLAB, you can represent time in a date string using either a 12-hour or 24-hour system in MATLAB. The following table shows how to create a 12-hour time string in the first column, and how to convert that time to its 24-hour equivalent:

12-hour time	Command to convert to 24-hour format	Equivalent 24-hour time
05:32 AM	`datestr('05:32 AM', 'HH:MM')`	5:32
05:32 PM	`datestr('05:32 PM', 'HH:MM')`	17:32

The next table shows how to create a 24-hour time string in the first column, and how to convert that time to its 12-hour equivalent.

24-hour time	Command to convert to 12-hour format	Equivalent 12-hour time
05:32	datestr('05:32', 'HH:MM PM')	5:32 AM
17:32	datestr('17:32', 'HH:MM PM')	5:32 PM

Warning   The terms AM and PM, when used in the field specifier string, can be misleading. These terms do not influence which characters actually become part of the date string; they only determine whether or not to include them in the date string. MATLAB selects AM versus PM based on the time entered.

A few other things to remember when specifying time in MATLAB are:

• When you use AM or PM, the HH field is also required .

• When you do not use AM and PM, single-digit hours display a leading zero .

Date String Tables

Symbolic Identifiers for Individual Fields
The following table shows:

• The nine fields of a date string (left column)

• Ways to format each field

• Example output for each field

• Assigned character-based field identifiers

Field	String Format	Example of Output	Field Identifier
Quarter year	Letter Q and 1 digit	Q4	`'QQ'`
Year	4 digits	2007	`'yyyy'`
	2 digits	07	`'yy'`
Month	Full name	December	'mmmm'
	First 3 letters	Dec	'mmm'
	2 digits	12	'mm'
	First letter	D	'm'
Day	Full name	Tuesday	`'dddd'`
	First 3 letters	Tue	`'ddd'`
	2 digits	20	`'dd'`
	First letter	T	`'d'`
Hour	2 digits	27	`'HH'`
Minute	2 digits	02	`'MM'`
Second	2 digits	59	`'SS'`
Millisecond	Decimal point and 3 digits	.057	`'FFF'`
12-hour period	AM or PM	PM	`'AM'` or `'PM'`

Notes Concerning Usage. Here are a few points to remember when using the symbolic identifiers:

MATLAB interprets the field specifiers in this table according to your computer's language setting and the current MATLAB language setting.

In a field specifier string, you cannot specify any field more than once. For example, you cannot use `'yy-mmm-dd-m'` because it has two month identifiers. The one exception to this is that you can combine one instance of dd with one instance of any of the other day identifiers:

```
ds = datestr(now, 'dddd  mmm dd yyyy')
ds =
    Wednesday  Jun 30 2010
```

You only can use QQ (quarter of the year) alone or with a year specifier.

Numeric Identifiers for Predefined Formats
The following table shows numeric identifiers you can use to include certain field and format combinations in a date string.

Date String Formats	Example of Output	Identifier to Use
'dd-mmm-yyyy HH:MM:SS'	01-Mar-2000 15:45:17	0
'dd-mmm-yyyy'	01-Mar-2000	1
'mm/dd/yy'	03/01/00	2
'mmm'	Mar	3
'm'	M	4
'mm'	03	5
'mm/dd'	03/01	6
'dd'	01	7
'ddd'	Wed	8
'd'	W	9
'yyyy'	2000	10
'yy'	00	11
'mmmyy'	Mar00	12
'HH:MM:SS'	15:45:17	13
'HH:MM:SS PM'	3:45:17 PM	14

Date String Formats	Example of Output	Identifier to Use
`'HH:MM'`	`15:45`	15
`'HH:MM PM'`	`3:45 PM`	16
`'QQ-YY'`	`Q1-01`	17
`'QQ'`	`Q1`	18
`'dd/mm'`	`01/03`	19
`'dd/mm/yy'`	`01/03/00`	20
`'mmm.dd,yyyy HH:MM:SS'`	`Mar.01,2000 15:45:17`	21
`'mmm.dd,yyyy'`	`Mar.01,2000`	22
`'mm/dd/yyyy'`	`03/01/2000`	23
`'dd/mm/yyyy'`	`01/03/2000`	24
`'yy/mm/dd'`	`00/03/01`	25
`'yyyy/mm/dd'`	`2000/03/01`	26
`'QQ-YYYY'`	`Q1-2001`	27
`'mmmyyyy'`	`Mar2000`	28
`'yyyy-mm-dd'` (ISO 8601)	`2000-03-01`	29
`'yyyymmddTHHMMSS'` (ISO 8601)	`20000301T154517`	30
`'yyyy-mm-dd HH:MM:SS'`	`2000-03-01 15:45:17`	31

Working with Date Vectors

Date vectors are an internal format for some MATLAB functions. You do not typically use date vectors in calculations, although you can use them to perform some simple computations such as the one shown in the example in this section.

A date vector is a 1-by-6 matrix of double-precision numbers arranged in the following order.

```
year month day hour minute second
```

The following date vector represents 10:45:07 AM on October 24, 2009.

```
[2009  10  24  10  45  07]
```

The fields of a date vector must follow these guidelines:

- All six elements, or *fields*, of the vector are required. If you are interested only in the date, and not the time, you can set the last three digits of the vector to zero.

- Time values are expressed in 24-hour notation. There is no AM or PM setting.

- The values for any of the six fields must be within an approximate range of 300 greater than or 550 less than the current value for that field.

Creating a Date Vector

As with any vector, you can create a date vector just as shown here. Be sure to put the fields in the correct order:

```
dv = [2010 8 23 16 35 42];
```

You can also create a date vector by converting a date string or serial date number. The `datevec` function converts from a date string or serial date number to a date vector.

Converting from Date String to Date Vector

The first argument to `datevec` can be a date string. If this string is in the default format for a date string, then you need only the one input argument:

```
dv = datevec('23-Aug 2010 16:35')
dv =
    2010     8    23    16    35     0
```

If the date string is in a nondefault date string format such as the one shown below, it is recommended that you also pass a field specifier argument. This argument tells MATLAB how the input string has been formatted:

```
dv = datevec('August 23, 2010 16:35', 'mmmm dd, yyyy HH:MM')
```

```
dv =
      2010        8       23       16       35        0
```

The tables "Symbolic Identifiers for Individual Fields" on page 2-58 and "Numeric Identifiers for Predefined Formats" on page 2-60 list all of the format specifiers for date strings. When converting from a date string to a date vector, you can use any string from these tables except for those that include the letter Q in the string (e.g., 'QQ-YYYY').

Converting from Serial Date Number to Date Vector

To convert from a serial date number to a date vector, pass only the date number argument:

```
dv = datevec(7.343736909722223e+005)
dv =
      2010        8       23       16       35        0
```

This argument can be the output of a function that yields a serial date number such as the now function:

```
dv = datevec(now)
dv =
  1.0e+003 *
    2.0100    0.0060    0.0290    0.0100    0.0350    0.0460
```

Creating Multiple Date Vectors

Calling datevec with more than one date string input returns a character array of converted date strings. Pass the multiple date strings in a cell array. All input date strings must use the same format. For example, the following command passes three dates that all use the mm/dd/yyyy format:

```
datevec({'09/16/2007';'05/14/1996';'11/29/2010'})
ans =
      2007        9       16        0        0        0
      1996        5       14        0        0        0
      2010       11       29        0        0        0
```

Milliseconds in Serial Date Numbers

Date vectors have no separate field in which to specify milliseconds. However, the seconds field has a fractional part and accurately keeps the milliseconds field. This example converts a date string with 647 milliseconds into a vector, and then converts it back into a string. Note that MATLAB fully restores the milliseconds count:

```
datenum([2010      8      23      16      35      0])
ans =
   7.3437e+005
dvec = datevec('11:21:02.647', 'HH:MM:SS.FFF')
dvec =
   1.0e+003 *
     2.0100    0.0010    0.0010    0.0110    0.0210    0.0026

datestr(dvec, 'HH:MM:SS.FFF')
ans =
    11:21:02.647
```

Examples of Using Date Vectors

Distribute the three time values in this vector to separate output variables:

```
[~, ~, ~, hour, min, sec] = datevec(now)
hour =
     11
min =
     47
sec =
    19.4660
```

Construction of the Eiffel Tower was begun on January 26 in 1887 and completed on March 31, 1889. This example finds the difference between the year, month, and day elements of the starting and ending date vectors to compute the time the tower was under construction:

```
startDate  = [1887  1  26  0  0  0];        % January 26, 1887
finishDate = [1889  3  31  0  0  0];        % March 31, 1889

t = finishDate - startDate
t =
```

```
      2     2     5     0     0     0

[fprintf('\nThe Eiffel Tower took %d years, ', t(1)) ...
 fprintf('%d months, and %d days to construct.\n', t(1), t(2))];

The Eiffel Tower took 2 years, 2 months, and 2 days to construct.
```

Working with Serial Date Numbers

A serial date number represents a calendar date as the number of days that has passed since a fixed base date.

In MATLAB, serial date number 1 is January 1, 0000. MATLAB also uses serial time to represent fractions of days beginning at midnight; for example, 6 p.m. equals 0.75 serial days. So the string '31-Oct-2003, 6:00 PM' in MATLAB is date number 731885.75.

MATLAB works internally with serial date numbers. If you are using functions that handle large numbers of dates or doing extensive calculations with dates, you get better performance if you use date numbers.

You can create a serial date number from a date string or date vector using the following commands:

Converting from Date String to Serial Date Number

The first argument to datenum can be a date string. If this string is in the default format for a date string, then you need only the one input argument:

```
dv = datenum('23-Aug 2010 16:35')
dn =
    7.3437e+005
```

If the date string is in a nondefault date string format such as the one shown below, it is recommended that you also pass a field specifier argument. The field specifier tells MATLAB how the input string has been formatted:

```
dn = datenum('August 23, 2010 16:35', 'mmmm dd, yyyy HH:MM')
dn =
  7.3437e+005
```

The tables "Symbolic Identifiers for Individual Fields" on page 2-58 and "Numeric Identifiers for Predefined Formats" on page 2-60 list all of the format specifiers for date strings. When converting from a date string to a serial date number, you can use any string from these tables except for those that include the letter Q in the string (e.g., 'QQ-YYYY').

Certain formats may not contain enough information to compute a date number. In these cases, hours, minutes, seconds, and milliseconds default to 0, the month defaults to January, the day to 1, and the year to the current year.

Converting from Date Vector to Serial Date Number

To convert from a date vector to a serial date number, pass just the date vector argument:

```
datenum([2010 8 23 16 35 42])
ans =
    7.3437e+005
```

This argument can be the output of a function that yields a date vector such as the clock function:

```
dv = clock
dv =
  1.0e+003 *
    2.0100    0.0060    0.0290    0.0110    0.0130    0.0057

datenum(dv)
ans =
  7.3432e+005
```

Creating Multiple Serial Date Numbers

Calling datenum with more than one date string input returns a character array of converted date strings. Pass the multiple date strings in a cell array. All input date strings must use the same format. For example, the following command passes three dates that all use the mm/dd/yyyy format:

```
datenum({'09/16/2007';'05/14/1996';'11/29/2010'})
```

```
ans =
     733301
     729159
     734471
```

Examples of Using Serial Date Numbers

Add 50 days to the serial date number returned by the now function:

```
fprintf('\n   In 50 days it will be %s.\n', ...
datestr(now+50, 'dddd, mmmm dd'))

   In 50 days it will be Saturday, July 10.
```

Using the addtodate function, add 20 days to the date 20-Jan-2002. This date is first converted to a date number by a nested call to datenum:

```
addtodate(datenum('20.01.2002','dd.mm.yyyy'),20,'day')
R =
     731256
```

Other Considerations

- "Carrying to the Next Field" on page 2-67
- "Specifying a Pivot Year" on page 2-68
- "Date Vectors vs. Vectors of Date Numbers" on page 2-68

Carrying to the Next Field

Where reasonable, MATLAB automatically carried values outside the normal range of each unit to the next field. For example, specifying a date vector with a month value of 14 affects the output by setting the month to February and incrementing the year value by 1. The carrying forward of values applies only to date vectors and to time and day values in date strings, carries the 10 extra months from the input date string 22/03/2009 into October of the following year:

```
datestr([2009 22 03 00 00 00])
ans =
     03-Oct-2010
```

All units can wrap and have negative values. Note that specifying a negative day value, `D`, sets the output to the last day of the previous month minus `|D|`. This example takes the input month (07, or July), finds the last day of the previous month (June 30), and subtracts the number of days in the field specifier (5 days) from that date to yield a return date of June 25, 2010:

```
datestr([2010 07 -05 00 00 00])
ans =
    25-Jun-2010
```

Specifying a Pivot Year

Use the pivot year to interpret date strings that specify the year using two characters. The pivot year is the starting year of the 100-year range in which a two-character date string year resides. The default pivot year is the current year minus 50 years.

The syntax for entering a pivot year is:

```
newString = datestr(oldString, fieldSpec, pivotYear);
```

This example changes the pivot year. Note the effect on the output:

```
datestr('4/16/55', 1, 1900)
ans =
    16-Apr-1955

datestr('4/16/55', 1, 2000)
ans =
    16-Apr-2055
```

Date Vectors vs. Vectors of Date Numbers

A six-element vector could represent either a single date vector, or a vector of six individual serial date numbers. For example, the vector [2010 12 15 11 45 03] could represent either 11:45:03 on December 15, 2010 or a vector of serial date numbers 2010, 12, 15, and so on.

If you are working with a date vector that the date conversion functions interpret by default as a vector of serial date numbers, you might need to explicitly convert your vector to a date number first, and then convert the value returned by datenum to the desired format.

In this example, the year 3000 is beyond the range of years for which MATLAB defaults to date vector format. Because of this, the intended format of the input vector is unclear to the datestr function, and the input is considered to be a vector of date numbers:

```
datestr([3000 11 05 10 32 56])
ans =
    18-Mar-0008
    11-Jan-0000
    05-Jan-0000
    10-Jan-0000
    01-Feb-0000
    25-Feb-0000
```

If it is your intention that the input is a date vector instead, convert it to a date number first, and then to a date string:

```
dn = datenum([3000 11 05 10 32 56]);
ds = datestr(dn)
ds =
    05-Nov-3000 10:32:56
```

Function Summary

MATLAB provides the following functions for time and date handling.

Current Date and Time Functions

Function	Description
clock	Return the current date and time as a date vector
date	Return the current date as date string
now	Return the current date and time as serial date number

Conversion Functions

Function	Description
datenum	Convert to a serial date number
datestr	Convert to a string representation of the date
datevec	Convert to a date vector

Utility Functions

Function	Description
addtodate	Modify a date number by field
calendar	Return a matrix representing a calendar
datetick	Label axis tick lines with dates
eomday	Return the last day of a year and month
weekday	Return the current day of the week

Timing Measurement Functions

Function	Description
cputime	Return the total CPU time used by MATLAB since it was started
etime	Return the time elapsed between two date vectors
tic, toc	Measure the time elapsed between invoking tic and toc

Regular Expressions

Overview

A regular expression is a string of characters that defines a certain pattern. You normally use a regular expression to search text for a group of words that matches the pattern. for example, while parsing program input or while processing a block of text.

The string `'Joh?n\w*'` is an example of a regular expression. It defines a pattern that starts with the letters `Jo`, is optionally followed by the letter `h` (indicated by `'h?'`), is then followed by the letter `n`, and ends with any number of *word characters* [1] (indicated by `'\w*'`). This pattern matches any of the following:

```
Jon, John, Jonathan, Johnny
```

Regular expressions provide a unique way to search a volume of text for a particular subset of characters within that text. Instead of looking for an exact character match as you would do with a function like `strfind`, regular expressions give you the ability to look for a particular *pattern* of characters.

For example, several ways of expressing a metric rate of speed are:

```
km/h
km/hr
km/hour
kilometers/hour
kilometers per hour
```

You could locate any of the above terms in your text by issuing five separate search commands:

```
strfind(text, 'km/h');
strfind(text, 'km/hour');
     - etc. -
```

---

1. The term "word characters" in this text refers to characters that are alphabetic, numeric, or underscore.

To be more efficient, however, you can build a single phrase that applies to all of these search strings:

```
'k' or 'kilo'  followed by ...
        'm' or 'meters'  followed by ...
                '/' or 'per'  followed by ...
                        'h' or 'hr' or 'hour'
```
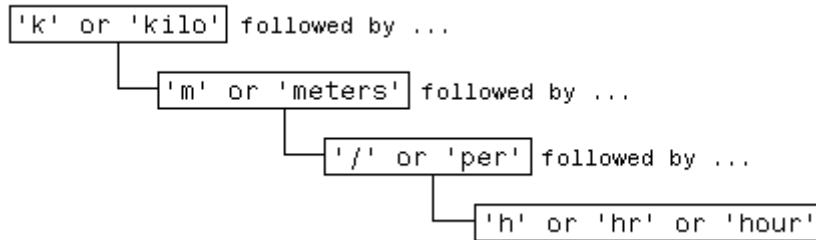
Translate this phrase it into a regular expression (to be explained later in this section) and you have:

```
pattern = 'k(ilo)?m(eters)?(/|\sper\s)h(r|our)?';
```

Now locate one or more of the strings using just a single command:

```
text = ['The high-speed train traveled at 250 ', ...
    'kilometers per hour alongside the automobile ', ...
    'travelling at 120 km/h.'];

regexp(text, pattern, 'match')
ans =
    'kilometers per hour'    'km/h'
```

Calling Regular Expression Functions from MATLAB

This section covers the following topics:

> **Note** The examples in this and some of the later sections of this documentation use expressions that can be difficult to decipher for anyone not previously exposed to them. The purpose of these initial examples is to introduce the basic use of regular expressions in MATLAB. Learning how to translate the expressions begins in the "Metacharacters and Operators" on page 2-82 section.

MATLAB Regular Expression Functions

There are four MATLAB functions that support searching and replacing characters using regular expressions. The first three are similar in the input values they accept and the output values they return. For details, click the links in the table to see the corresponding function reference pages in the MATLAB Help.

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.
regexptranslate	Translate string into regular expression.

When calling any of the first three functions, pass the string to be parsed and the regular expression in the first two input arguments. When calling regexprep, pass an additional input that is an expression that specifies a pattern for the replacement string.

Returning the Desired Information

The regexp and regexpi functions return from 1 to 7 output values. providing the following information:

- The content or array indices of all matching strings

- The content of all nonmatching strings

- The content, names, or array indices of all tokens that were found

Unless you specify otherwise, MATLAB returns as many output values as you have output variables for. These are returned in the order shown by the "Return Value Summary" on page 2-133 table.

The following call to regexp returns all 7 outputs:

```
[matchStart, matchEnd, tokenIndices, matchStrings, ...
    tokenStrings, tokenName, splitStrings] = regexp(str, expr);
```

To specify fewer values to return, include an identifying keyword in the input argument list when you call regexp or regexpi. For example, the following statement uses two of these keywords, match and start:

```
[matchStrings, matchStart] = regexp(str, expr, 'match', 'start')
```

When you execute this statement, MATLAB assigns a cell array of all strings that match the pattern to variable matchStrings, and assigns an array of doubles containing the starting index of each match to variable matchStart.

For information on these output values and selecting which outputs to return, see the regexp function reference page.

Modifying Parameters of the Search

You can fine-tune your regular expression parsing using the optional mode inputs: Case Sensitivity, Dot Matching, Anchor Type, and Spacing. These modes tell MATLAB whether or not to:

- Consider letter case when matching an expression to a string (Case Sensitivity mode).

- Include the newline (\n) character when matching the dot (.) metacharacter in a regular expression (Dot Matching mode).

- Consider the ^ and $ metacharacters to represent the beginning and end of a string or the beginning and end of a line (Anchor Type mode).

- Ignore space characters and comments in the expression or to interpret them literally (Spacing mode).

Applying Modes. You can apply any of these modes in either of two ways:

- Apply the mode to *all* of a regular expression by passing the mode specifier in the argument list of the call. See Example 1, below.

- Apply the mode to *specific parts* of your expression by specifying the mode symbolically within the regular expression itself. See Example 2, below.

Example 1 — Applying Case Sensitivity Mode to the Entire String.

Create two slightly different strings, s1 and s2. Then write an expression expr that you can use to match both of these strings, but only when ignoring case. (The expression operators .+ match any consecutive series of any character between the MAT or mat phrases.)

```
s1 = 'Save your MATLAB data to a .mat file in C:\work\matlab';
s2 = 'Save your MATLAB data to a .MAT file in C:\work\matlab';
expr = '.+MAT.+mat.+mat.+';
```

Run regexp on both strings at the same time in ignorecase mode and examine the output in cell array c:

```
c = regexp({s1, s2}, expr, 'match', 'ignorecase');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\matlab'
ans =
    'Save your MATLAB data to a .MAT file in C:\work\matlab'
```

Because of the ignorecase mode, there is a match for both strings. When you use matchcase mode instead, only the exact case match is accepted:

```
c = regexp({s1, s2}, expr, 'match', 'matchcase');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\matlab'
ans =
     {}
```

Example 2 — Applying Case Sensitivity Mode Selectively. This example uses symbolic mode designators within the expression itself. The (?i) symbol tells regexp to ignore case for that part of the expression that immediately follows it. Similarly, the (?-i) symbol requires case to match for the part of the expression following it.

Here are three strings that vary slightly in case. Following that is the expression expr that employs the two states of the Case Sensitivity mode. Note that each of the (?-i) or (?i) symbols used in this expression applies only to the letters MAT or mat that immediately follow it:

```
s1 = 'Save your MATLAB data to a .mat file in C:\work\matlab';
s2 = 'Save your MATLAB data to a .MAT file in C:\work\MATLAB';
s3 = 'Save your MATLAB data to a .MAT file in C:\work\matlab';
expr = '.*(?-i)MAT.*(?i)mat.*(?-i)mat';
```

Run regexp on the three strings. According to the expression expr, the first and third instances of the letters 'mat' must be in upper and lower case, respectively. Case is ignored for the second instance. Only strings s1 and s3 satisfy this condition:

```
c = regexp({s1,s2,s3}, expr, 'match');
c{:}
ans =
    'Save your MATLAB data to a .mat file in C:\work\mat'
ans =
     {}
ans =
    'Save your MATLAB data to a .MAT file in C:\work\mat'
```

Parsing Strings with Regular Expressions

MATLAB parses a string from left to right, "consuming" the string as it goes. If matching characters are found, regexp records the location and resumes parsing the string, starting just after the end of the most recent match. There is no overlapping of characters in this process. See Examples 2a and 2b under "Using the Lookahead Operator" if you need to match overlapping character groups.

There are three steps involved in using regular expressions to search text for a particular string:

1 Identify unique patterns in the string

   This entails breaking up the string you want to search for into groups of like character types. These character types could be a series of lowercase letters, a dollar sign followed by three numbers and then a decimal point, etc.

2 Express each pattern as a regular expression

   Use the *metacharacters* and operators described in this documentation to express each segment of your search string as a regular expression. Then combine these expression segments into the single expression to use in the search.

3 Call the appropriate search function

   Pass the string you want to parse to one of the search functions, such as `regexp` or `regexpi`, or to the string replacement function, `regexprep`.

The example shown in this section searches a record containing contact information belonging to a group of five friends. This information includes each person's name, telephone number, place of residence, and e-mail address. The goal is to extract specific information from one or more of the strings.

```
contacts = { ...
'Harry  287-625-7315  Columbus, OH  hparker@hmail.com'; ...
'Janice  529-882-1759  Fresno, CA  jan_stephens@horizon.net'; ...
'Mike  793-136-0975  Richmond, VA  sue_and_mike@hmail.net'; ...
'Nadine  648-427-9947  Tampa, FL  nadine_berry@horizon.net'; ...
'Jason  697-336-7728  Montrose, CO  jason_blake@mymail.com'};
```

The first part of the example builds a regular expression that represents the format of a standard e-mail address. Using that expression, the example then searches the information for the e-mail address of one of the group of friends. Contact information for Janice is in row 2 of the `contacts` cell array:

```
contacts{2}
ans =
   Janice  793-882-1759  Fresno, CA  jan_stephens@horizon.net
```

Step 1 — Identify Unique Patterns in the String

A typical e-mail address is made up of standard components: the user's account name, followed by an @ sign, the name of the user's internet service provider (ISP), a dot (period), and the domain to which the ISP belongs. The table below lists these components in the left column, and generalizes the format of each component in the right column.

Unique patterns of an email address	General description of each pattern
Start with the account name `jan_stephens ...`	One or more lowercase letters and underscores
Add '@' `jan_stephens@ ...`	`@` sign
Add the ISP `jan_stephens@horizon ...`	One or more lowercase letters, no underscores
Add a dot (period) `jan_stephens@horizon. ...`	Dot (period) character
Finish with the domain `jan_stephens@horizon.net`	`com` or `net`

Step 2 — Express Each Pattern as a Regular Expression

In this step, you translate the general formats derived in Step 1 into segments of a regular expression. You then add these segments together to form the entire expression.

The table below shows the generalized format descriptions of each character pattern in the left-most column. (This was carried forward from the right column of the table in Step 1.) The second column links to tables in this documentation that show the appropriate expressions to use in translating this description into a regular expression. The third column shows the operators or metacharacters chosen from those tables to represent the character pattern.

Description of each segment	Tables referenced	Related metacharacters
One or more lowercase letters and underscores	See Character Types on page 2-127, Quantifiers on page 2-129.	`[a-z_]+`
@ sign	See Character Representation on page 2-127.	`@`
One or more lowercase letters, no underscores	See Character Types on page 2-127, Quantifiers on page 2-129.	`[a-z]+`
Dot (period) character	See Character Representation on page 2-127.	`\.`
`com` or `net`	See Grouping Operators on page 2-128.	`(com|net)`

Assembling these metacharacters into one string gives you the complete expression:

```
email = '[a-z_]+@[a-z]+\.(com|net)';
```

Step 3 — Call the Appropriate Search Function

In this step, you use the regular expression derived in Step 2 to match an e-mail address for one of the friends in the group. Use the regexp function to perform the search.

Here is the list of contact information shown earlier in this section. Each person's record occupies a row of the contacts cell array:

```
contacts = { ...
'Harry  287-625-7315  Columbus, OH  hparker@hmail.com'; ...
'Janice  529-882-1759  Fresno, CA  jan_stephens@horizon.net'; ...
'Mike  793-136-0975  Richmond, VA  sue_and_mike@hmail.net'; ...
'Nadine  648-427-9947  Tampa, FL  nadine_berry@horizon.net'; ...
'Jason  697-336-7728  Montrose, CO  jason_blake@mymail.com'};
```

This is the regular expression that represents an e-mail address, as derived in Step 2:

```
email = '[a-z_]+@[a-z]+\.(com|net)';
```

Call the regexp function, passing row 2 of the contacts cell array and the email regular expression. This returns the e-mail address for Janice.

```
regexp(contacts{2}, email, 'match')
ans =
    'jan_stephens@horizon.net'
```

Note The last input passed to regexp in this command is the keyword 'match' This keyword causes regexp to return the output as a string instead of as indices into the cell array.

Make the same call, but this time for the fifth person in the list:

```
regexp(contacts{5}, email, 'match')
ans =
    'jason_blake@mymail.com'
```

You can also search for the e-mail address of everyone in the list by using the entire cell array for the input string argument:

```
regexp(contacts, email, 'match');
```

Other Benefits of Using Regular Expressions

In addition to parsing single strings, you can also use the MATLAB regular expression functions for any of the following tasks:

- "Parsing or Replacing with Multiple Expressions and Strings" on page 2-81
- "Replacing Parts of a String" on page 2-82
- "Matching with Tokens Taken from the String" on page 2-82
- "Matching and Replacing Strings Dynamically" on page 2-82

Parsing or Replacing with Multiple Expressions and Strings

The MATLAB regular expression functions also work on multiple strings contained in a cell array. You can use multiple strings as the strings to be parsed, as regular expressions to match against the parse string(s), as replacement strings, or most combinations of these.

Replacing Parts of a String

String replacement with regular expressions requires the `regexprep` function. This function accepts two regular expressions in its input argument list. Each expression specifies a character pattern to match in the string to be parsed. The function then replaces occurrences of the first pattern with occurrences of the second.

Matching with Tokens Taken from the String

A token is one or more characters selected from within the string being parsed that you can use to match other characters in the same string. The characters representing a token are not constants; they depend upon the contents of the parse string that match a part of the expression. You define a token by enclosing part of a regular expression in parentheses. You search for that token using the metacharacters \1, \2, etc. You can also use tokens in specifying a replacement string for the `regexprep` function. In this case, you refer to specific tokens using the metacharacters $1, $2, etc.

Matching and Replacing Strings Dynamically

With dynamic expressions, you can:

- Execute a MATLAB command within your expression parsing command.

- Execute a MATLAB command, and include the returned string in the match expression.

- Parse a regular expression and include the resulting string in the match expression.

Metacharacters and Operators

Much of the remainder of this section on regular expressions documents the various metacharacters and operators that you need to compose your expressions.

Category	Metacharacters and Operators
"Character Type Operators" on page 2-84	One of a certain group of characters (e.g., a character in a predefined set or range, a whitespace character, an alphabetic, numeric, or underscore character, or a character that is not in one of these groups.
"Character Representation" on page 2-88	Metacharacters that represent a special character (e.g., backslash, new line, tab, hexadecimal values, any untranslated literal character, etc.
"Grouping Operators" on page 2-89	A grouping of letters or metacharacters to apply a regular expression operator to.
"Nonmatching Operators" on page 2-91	Text included in an expression for the purpose of adding a comment statement, but not to be used as a pattern to find a match for.
"Positional Operators" on page 2-92	Location in the string where the characters or pattern must be positioned for there to be a match (e.g., start or end of the string, start or end of a word, an entire word).
"Lookaround Operators" on page 2-93	Characters or patterns that immediately precede or follow the intended match, but are not considered to be part of the match itself.
"Quantifiers" on page 2-99	Various ways of expressing the number of times a character or pattern is to occur for there to be a match (e.g., exact number, minimum, maximum, zero or one, zero or more, one or more, etc.)
"Tokens" on page 2-102	Characters or patterns selected from the string being parsed that you can use to match other characters in the string.
"Named Capture" on page 2-107	Operators used in assigning names to matched tokens, thus making your code more maintainable and the output easier to interpret.
"Conditional Expressions" on page 2-109	Operators that express conditions under which a certain match is considered to be is acceptable.

Category	Metacharacters and Operators
"Dynamic Regular Expressions" on page 2-111	Operators that include a subexpression or command that MATLAB parses or executes. MATLAB uses the result of that operation in parsing the overall expression.
"String Replacement" on page 2-120	Operators used with the `regexprep` function to specify the content of the replacement text.

Character Type Operators

Tables and examples in this and subsequent sections show the operators and syntax supported by the MATLAB `regexp`, `regexpi`, and `regexprep` functions. Expressions shown in the left column have special meaning and match one or more characters according to the usage described in the right column. Any character not having a special meaning, for example, any alphabetic character, matches that same character literally. To force one of the regular expression functions to interpret a sequence of characters literally (rather than as an operator) use the `regexptranslate` function.

Character types represent either a specific set of characters (e.g., uppercase) or a certain type of character (e.g., nonwhitespace).

Operator	Usage
`.`	Any single character, including white space
`[`$c_1 c_2 c_3$`]`	Any character contained within the brackets: c_1 or c_2 or c_3
`[^`$c_1 c_2 c_3$`]`	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
`[`c_1`-`c_2`]`	Any character in the range of c_1 through c_2
`\s`	Any white-space character; equivalent to `[ \f\n\r\t\v]`
`\S`	Any nonwhitespace character; equivalent to `[^ \f\n\r\t\v]`

Operator	Usage
\w	Any alphabetic, numeric, or underscore character. For English character sets, this is equivalent to `[a-zA-Z_0-9]`.
\W	Any character that is not alphabetic, numeric, or underscore. For English character sets, this is equivalent to `[^a-zA-Z_0-9]`.
\d	Any numeric digit; equivalent to `[0-9]`
\D	Any nondigit character; equivalent to `[^0-9]`

The following examples demonstrate how to use the character classes listed above. See the `regexp` reference page for help with syntax. Most of these examples use the following string:

```
str = 'The rain in Spain falls mainly on the plain.';
```

Any Character — .

The `.` operator matches any single character, including whitespace.

Example 1 — Matching Any Character. Use the dot (.) operator to locate sequences of five consecutive characters that end with `'ain'`. The regular expression used in this example is

```
expr = '..ain';
```

Find each occurrence of the expression `expr` within the input string `str`. Return a vector of the indices at which any matches begin:

```
str = 'The rain in Spain falls mainly on the plain.';

startIndex = regexp(str, expr)
startIndex =
     4    13    24    39
```

Here is the input string with the returned `startIndex` values shown below it. Note that the dot operator not only matches the letters in the string, but whitespace characters as well:

```
The rain in Spain falls mainly on the plain.
   |      |          |            |
   4      13         24           39
```

If you would prefer to have MATLAB return the text of the matching substrings, use the `'match'` qualifier in the command:

```
matchStr = regexp(str, expr, 'match')
matchStr =
    ' rain'    'Spain'    ' main'    'plain'
```

Example 2 — Returning Strings Rather than Indices. Here is the same example, this time specifying the command qualifier `'match'`. In this case, regexp returns the *text* of the matching strings rather than the starting index:

```
regexp(str, '..ain', 'match')
ans =
    ' rain'    'Spain'    ' main'    'plain'
```

Selected Characters — [c1c2c3]

Use $[c_1c_2c_3]$ in an expression to match selected characters r, p, or m followed by `'ain'`. Specify two qualifiers this time, `'match'` and `'start'`, along with an output argument for each, mat and idx. This returns the matching strings and the starting indices of those strings:

```
[mat idx] = regexp(str, '[rpm]ain', 'match', 'start')
mat =
    'rain'    'pain'    'main'
idx =
     5    14    25
```

Range of Characters — [c1 - c2]

Use $[c_1\text{-}c_2]$ in an expression to find words that begin with a letter in the range of A through Z:

```
[mat idx] = regexp(str, '[A-Z]\w*', 'match', 'start')
mat =
    'The'    'Spain'
idx =
     1    13
```

Word and White-Space Characters — \w, \s

Use \w and \s in an expression to find words that end with the letter n followed by a white-space character. Add a new qualifier, 'end', to return the str index that marks the end of each match:

```
[mat ix1 ix2] = regexp(str, '\w*n\s', 'match', 'start', 'end')
mat =
    'rain '    'in '    'Spain '    'on '
ix1 =
     5    10    13    32
ix2 =
     9    12    18    34
```

Numeric Digits — \d

Use \d to find numeric digits in the following string:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\d', 'match', 'start')
mat =
    '1'    '2'    '3'
idx =
     9    12    15
```

Character Representation

The following character combinations represent specific character and numeric values.

Operator	Usage
\a	Alarm (beep)
\\	Backslash
\$	Dollar sign
\b	Backspace
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\oN or \o{N}	Character of octal value N
\xN or \x{N}	Character of hexadecimal value N
\char	If a character has special meaning in a regular expression, precede it with backslash (\) to match it literally.

Octal and Hexadecimal — \o, \x

Use \x and \o in an expression to find a comma (hex 2C) followed by a space (octal 40) followed by the character 2:

```
numstr = 'Easy as 1, 2, 3';

[mat idx] = regexp(numstr, '\x2C\o{40}2', 'match', 'start')
mat =
    ', 2'
idx =
    10
```

Grouping Operators

When you need to use one of the regular expression operators on a number of consecutive elements in an expression, group these elements together with one of the grouping operators and apply the operation to the entire group. For example, this command matches a capital letter followed by a numeral and then an optional space character. These elements have to occur at least two times in succession for there to be a match. To apply the `{2,}` multiplier to all three consecutive characters, you can first make a group of the characters and then apply the `(?:)` quantifier to this group:

```
regexp('B5 A2 6F 63 R6 P4 B2 BC', '(?:[A-Z]\d\s?){2,}', 'match')
ans =
    'B5 A2 '    'R6 P4 B2 '
```

There are three types of explicit grouping operators that you can use when you need to apply an operation to more than just one element in an expression. Also in the grouping category is the alternative match (logical OR) operator, `|`. This creates two or more groups of elements in the expression and applies an operation to one of the groups.

Operator	Usage
`(expr)`	Group regular expressions and capture tokens.
`(?:expr)`	Group regular expressions, but do not capture tokens.
`(?>expr)`	Group atomically.
$expr_1$|$expr_2$	Match expression $expr_1$ or expression $expr_2$.

Grouping and Capture — (expr)

When you enclose an expression in parentheses, MATLAB not only treats all of the enclosed elements as a group, but also captures a token from these elements whenever a match with the input string is found. For an example of how to use this, see "Using Tokens — Example 1" on page 2-104.

Grouping Only — (?:expr)

Use `(?:expr)` to group a non-vowel (consonant, numeric, whitespace, punctuation, etc.) followed by a vowel in the palindrome `pstr`. Specify at least

two consecutive occurrences ({2,}) of this group. Return the starting and
ending indices of the matched substrings:

```
pstr = 'Marge lets Norah see Sharon''s telegram';
expr = '(?:[^aeiou][aeiou]){2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'Nora'    'haro'    'tele'
ix1 =
    12    23    31
ix2 =
    15    26    34
```

Remove the grouping, and the {2,} now applies only to [aeiou]. The
command is entirely different now as it looks for a non-vowel followed by at
least two consecutive vowels:

```
expr = '[^aeiou][aeiou]{2,}';

[mat ix1 ix2] = regexp(pstr, expr, 'match', 'start', 'end')
mat =
    'see'
ix1 =
    18
ix2 =
    20
```

Alternative Match — expr1|expr2

Use $p_1|p_2$ to pick out words in the string that start with let or tel:

```
regexpi(pstr, '(let|tel)\w+', 'match')
ans =
    'lets'    'telegram'
```

> **Note** The expressions A | B and B | A may return different answers. If there is a match with the first part of the expression (before the | symbol), then the second part (that follows the | symbol) is not considered.

See the following example. Both calls to `regexp` parse the same string, and, except for the order of the OR conditions, the same expression. But the first call returns two values and the second returns just one:

```
string = 'one two';    expr1 = '(\w+\s\w+)';    expr2 = '(\w+)';

regexp(string, [expr1 '|' expr2], 'match')
ans =
    'one two'

regexp(string, [expr2 '|' expr1], 'match')
ans =
    'one'     'two'
```

Nonmatching Operators

The `comment` operator enables you to insert comments into your code to make it more maintainable. The text of the comment is ignored by MATLAB when matching against the input string.

Operator	Usage
(?#comment)	Insert a comment into the expression. Comments are ignored in matching.

Including Comments — (?#expr)

Use `(?#expr)` to add a comment to this expression that matches capitalized words in `pstr`. Comments are ignored in the process of finding a match:

```
regexp(pstr, '(?# Match words in caps)[A-Z]\w+', 'match')
ans =
    'Marge'     'Norah'     'Sharon'
```

Positional Operators

Positional operators in an expression match parts of the input string not by content, but by where they occur in the string (e.g., the first N characters in the string).

Operator	Usage
^expr	Match expr if it occurs at the beginning of the input string.
expr$	Match expr if it occurs at the end of the input string.
\<expr	Match expr when it occurs at the beginning of a word.
expr\>	Match expr when it occurs at the end of a word.
\<expr\>	Match expr when it represents the entire word.

Start and End of String Match — ^expr, expr$

Use ^expr to match words starting with the letter m or M only when it begins the string, and expr$ to match words ending with m or M only when it ends the string:

```
regexpi(pstr, '^m\w*|\w*m$', 'match')
ans =
    'Marge'     'telegram'
```

Start and End of Word Match — \<expr, expr\>

Use \<expr to match any words starting with n or N, or ending with e or E:

```
regexpi(pstr, '\<n\w*|\w*e\>', 'match')
ans =
    'Marge'     'Norah'     'see'
```

Exact Word Match — \<expr\>

Use \<expr\> to match a word starting with an n or N and ending with an h or H:

```
regexpi(pstr, '\<n\w*h\>', 'match')
ans =
    'Norah'
```

Lookaround Operators

Lookaround operators tell MATLAB to look either ahead or behind the current location in the string for a specified expression. If the expression is found, MATLAB attempts to match a given pattern.

This table shows the four lookaround expressions: lookahead, negative lookahead, lookbehind, and negative lookbehind.

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.
(?!expr)	Look ahead from current position and test if expr is not found
(?<=expr)	Look behind from current position and test if expr is found.
(?<!expr)	Look behind from current position and test if expr is not found.

Lookaround operators do not change the current parsing location in the input string. They are more of a condition that must be satisfied for a match to occur.

For example, the following command uses an expression that matches alphabetic, numeric, or underscore characters (\w*) that meet the condition that they *look ahead to* (i.e., are immediately followed by) the letters vision. The resulting match includes only that part of the string that matches the \w* operator; it does not include those characters that match the lookahead expression (?=vision):

```
[s e] = regexp('telegraph television telephone', ...
               '\w*(?=vision)', 'start', 'end')
s =
    11
e =
```

```
      14
```

If you repeat this command and match one character beyond the lookahead expression, you can see that parsing of the input string resumes at the letter v, thus demonstrating that matching the lookahead operator has not consumed any characters in the string:

```
regexp('telegraph television telephone', ...
        '\w*(?=vision).', 'match')
ans =
     'telev'
```

Note You can also use lookaround operators to perform a logical AND of two elements. See "Using Lookaround as a Logical Operator" on page 2-98.

Using the Lookahead Operator — expr(?=test)

Example 1 — Lookahead. Look ahead to a file name (fileread.m), and return only the name of the folder in which it resides, not the file name itself. Note that the lookahead part of the expression serves only as a condition for the match; it is not part of the match itself:

```
str = which('fileread')
str =
   C:\Program Files\MATLAB\toolbox\matlab\iofun\fileread.m

% Look ahead to a backslash (\\), followed by a file name (\w+)
% with an .m or .p extension (\.[mp]). Capture the letters
% that precede this sequence.
regexp(str, '\w+(?=\\\w+\.[mp])', 'match')
ans =
     'iofun'
```

Example 2a — Matching Sequential Character Groups. MATLAB parses a string from left to right, "consuming" the string as it goes. If matching characters are found, regexp records the location and resumes parsing the string from the location of the most recent match. There is no overlapping of characters in this process.

Find all sequences of 6 nonwhitespace characters in the input string shown below. Following the MATLAB default behavior, do not allow for overlap. That is, begin looking for your next match starting just after the *end* of the current match:

```
string = 'Locate several 6-char. phrases';
regexpi(string, '\S{6}')
ans =
     1     8    16    24
```

This statement finds the phrases:

```
Locate   severa   6-char   phrase
```

Example 2b — Using Lookahead to Match Overlapping Character Groups. If you need to find *every* sequence of characters that match a pattern, including sequences that overlap another, capture only the first character and look ahead for the remainder of the pattern. In other words, begin looking for your next match starting after the *next character* of the current match:

```
string = 'Locate several 6-char. phrases';
regexpi(string, '\S(?=\S{5})')
ans =
     1     8     9    16    17    24    25
```

This statement finds the phrases:

```
Locate   severa   everal   6-char   -char.   phrase   hrases
```

Using the Negative Lookahead Operator — expr(?!test)

Example — Negative Lookbehind and Lookahead. Generate a series of sequential numbers:

```
n = num2str(5:15)
n =
   5   6   7   8   9  10  11  12  13  14  15
```

Use both the negative lookbehind and negative lookahead operators together to precede only the single-digit numbers with zero:

```
regexprep(n, '(?<!\d)(\d)(?!\d)', '0$1')
ans =
    05   06   07   08   09  10  11  12  13  14  15
```

Using the Lookbehind Operator — (?<=test)expr

Example 1 — Positive and Negative Lookbehind Operators. Using the lookbehind operator, find the letter r that is preceded by the letter u:

```
str = 'Neural Network Toolbox';

startIndex = regexp(str, '(?<=u)r', 'start')
startIndex =
     4
```

Using the negative lookbehind operator, find the letter r that is *not* preceded by the letter u:

```
startIndex = regexp(str, '(?<!u)r', 'start')
startIndex =
    13
```

Example 2 — Lookbehind. Return the names and 7-digit telephone numbers for those people in the list that are in the 617 area code. The lookbehind (?<=^617-) finds those lines that begin with the number 617:

```
phone_list = {...
'978-389-2457 Kevin';      '617-922-3091 Ruth'; ...
'781-147-1748 Alan';       '508-643-9648 George'; ...
'617-774-6642 Lisa';       '617-241-0275 Greg'; ...
'413-995-9114 Jason';      '781-276-0482 Victoria'};
len = length(phone_list);

ph617 = regexp(phone_list, '(?<=^617-).*', 'match');

for k=1:len
str = char(ph617{k});
if ~isempty(str),   fprintf('   %s\n', str),   end
end
```

MATLAB returns the three numbers that have a 617 area code:

```
922-3091 Ruth
774-6642 Lisa
241-0275 Greg
```

Using the Negative Lookbehind Operator— (?<!test)expr

Example — Negative Lookbehind. This example uses negative lookbehind to find those tasks that are not labelled as `Done` or `Pending`, Create a list of tasks, each with status information to the left:

```
tasks = {...
'ToDo    3892457';     'Done    9223091'; ...
'Pending 1471748';     'Maybe   7746642'; ...
'ToDo    2410275';     'Pending 4723596'; ...
'ToDo    9959114';     'Maybe   2760482'; ...
'ToDo    3080027';     'Done    1221941'};
count = length(tasks);
```

The regular expression looks for those task numbers that do not have a `Done` or `Pending` status. Note that you can use the or (|) operator in a lookaround to check for more than one condition:

```
doNow = regexp(tasks, '(?<!^(Done|Pending).*)\d+', 'match');
```

Now print out the results:

```
disp 'The following tasks need attention:'
for k=1:count
   s = char(doNow{k});
   if ~isempty(s),   fprintf('   %s\n', s),   end
end
```

The output displays all but the `Done` and `Pending` tasks:

```
The following tasks need attention:
   3892457
   7746642
```

```
        2410275
        9959114
        2760482
        3080027
```

Using Lookaround as a Logical Operator

One way in which a lookahead operation can be useful is to perform a logical AND between two conditions. This example initially attempts to locate all lowercase consonants in a text string. The text string is the first 50 characters of the help for the normest function:

```
helptext = help('normest');
str = helptext(1:50)
str =
 NORMEST Estimate the matrix 2-norm.
    NORMEST(S
```

Merely searching for non-vowels ([^aeiouAEIOU]) does not return the expected answer, as the output includes capital letters, space characters, and punctuation:

```
c = regexp(str, '[^aeiouAEIOU]', 'match')
c =
  Columns 1 through 12
    ' '  'N'  'R'  'M'  'S'  'T'  ' '  's'  't'  'm' 't'

       -- etc. --
```

Try this again, using a lookahead operator to create the following AND condition:

```
(lowercase letter) AND (not a vowel).
```

This time, the result is correct:

```
c = regexp(str, '(?=[a-z])[^aeiou]', 'match')
c =
  's'  't'  'm '  't'  't'  'h'  'm'  't'  'r'  'x'
     'n'  'r'  'm'
```

Note that when using a lookahead operator to perform an AND, you need to place the match expression expr *after* the test expression test:

```
(?=test)expr or (?!test)expr
```

Quantifiers

With the quantifiers shown below, you can specify how many instances of an element are to be matched. The basic quantifying operators are listed in the first six rows of the table.

By default, MATLAB matches as much of an expression as possible. Using the operators shown in the last two rows of the table, you can override this default behavior. Specify these options by appending a + or ? immediately following one of the six basic quantifying operators.

Operator	Usage
expr{m,n}	Must occur at least m times but no more than n times.
expr{m,}	Must occur at least m times.
expr{n}	Must match exactly n times. Equivalent to {n,n}.
expr?	Match the preceding element 0 times or 1 time. Equivalent to {0,1}.
expr*	Match the preceding element 0 or more times. Equivalent to {0,}.
expr+	Match the preceding element 1 or more times. Equivalent to {1,}.
q_expr+	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails. The term q_expr represents any of the expressions shown in the top six rows of this table.
q_expr?	Match only as much of the quantified expression as necessary. The term q_expr represents any of the expressions shown in the top six rows of this table. For an example, see "Lazy Quantifiers — expr*?" on page 2-101, below.

Zero or One — expr?

Use ? to make the HTML `<code>` and `</code>` tags optional in the string. The first string, hstr1, contains one occurrence of each tag. Since the expression uses ()? around the tags, one occurrence is a match:

```
hstr1 = '<td><a name="18854"></a><code>%%</code><br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr1, expr, 'match')
ans =
    '</a><code>%%</code><br>'
```

The second string, hstr2, does not contain the code tags at all. Just the same, the expression matches because ()? allows for zero occurrences of the tags:

```
hstr2 = '<td><a name="18854"></a>%%<br></td>';
expr = '</a>(<code>)?..(</code>)?<br>';

regexp(hstr2, expr, 'match')
ans =
    '</a>%%<br>'
```

Zero or More — expr*

The first regexp command looks for at least one occurrence of `<br>` and finds it. The second command parses a different string for at least one `<br>` and fails. The third command uses * to parse the same line for zero or more line breaks and this time succeeds.

```
hstr1 = '<p>This string has <br><br>line breaks</p>';
regexp(hstr1, '<p>.*(<br>).*</p>', 'match')
ans =
    '<p>This string has <br><br>line breaks</p>';

hstr2 = '<p>This string has no line breaks</p>';
regexp(hstr2, '<p>.*(<br>).*</p>', 'match')
ans =
    {}

regexp(hstr2, '<p>.*(<br>)*.*</p>', 'match')
```

```
ans =
    '<p>This string has no line breaks</p>';
```

One or More — expr+

Use + to verify that the HTML image source is not empty. This looks for one or more characters in the gif filename:

```
hstr = '<a href="s12.html"><img src="b_prev.gif" border=0>';
expr = '<img src="\w+.gif';

regexp(hstr, expr, 'match')
ans =
    '<img src="b_prev.gif'
```

Exact, Minimum, and Maximum Quantities — {min,max}

Use {m}, {m,}, and {m,n} to verify the href syntax used in HTML. This statement requires the href to have at least one nonwhitespace character, followed by exactly one occurrence of .html, optionally followed by # and five to eight digits:

```
hstr = '<a name="18749"></a><a href="s13.html#18760">';
expr = '<a href="\w{1,}(\.html){1}(\#\d{5,8}){0,1}"';

regexp(hstr, expr, 'match')
ans =
    '<a href="s13.html#18760"'
```

Lazy Quantifiers — expr*?

This example shows the difference between the default (*greedy*) quantifier and the *lazy* quantifier (?). The first part of the example uses the default quantifier to match all characters from the opening <tr to the ending </td:

```
hstr = '<tr valign=top><td><a name="19184"></a><br></td>';
regexp(hstr, '</?t.*>', 'match')
ans =
```

```
    '<tr valign=top><td><a name="19184"></a><br></td>'
```

The second part uses the lazy quantifier to match the minimum number of characters between `<tr`, `<td`, or `</td` tags:

```
regexp(hstr, '</?t.*?>', 'match')
ans =
    '<tr valign=top>'    '<td>'    '</td>'
```

Tokens

Parentheses used in a regular expression not only group elements of that expression together, but also designate any matches found for that group as *tokens.* You can use tokens to match other parts of the same string. One advantage of using tokens is that they remember what they matched, so you can recall and reuse matched text in the process of searching or replacing.

This section covers

- "Operators Used with Tokens" on page 2-102
- "Introduction to Using Tokens" on page 2-103
- "Using Tokens — Example 1" on page 2-104
- "Using Tokens — Example 2" on page 2-104
- "Tokens That Are Not Matched" on page 2-105
- "Using Tokens in a Replacement String" on page 2-107

Operators Used with Tokens

Here are the operators you can use with tokens in MATLAB.

Operator	Usage
(expr)	Capture in a token all characters matched by the expression within the parentheses.
\N	Match the Nth token generated by this command. That is, use \1 to match the first token, \2 to match the second, and so on.

Operator	Usage
$N	Insert the match for the Nth token in the replacement string. Used only by the regexprep function. If N is equal to zero, then insert the entire match in the replacement string.
(?(N)s1|s2)	If N[th] token is found, then match s1, else match s2

Introduction to Using Tokens

You can turn any pattern being matched into a token by enclosing the pattern in parentheses within the expression. For example, to create a token for a dollar amount, you could use '(\$\d+)'. Each token in the expression is assigned a number, starting from 1, going from left to right. To make a reference to a token later in the expression, refer to it using a backslash followed by the token number. For example, when referencing a token generated by the third set of parentheses in the expression, use \3.

As a simple example, if you wanted to search for identical sequential letters in a string, you could capture the first letter as a token and then search for a matching character immediately afterwards. In the expression shown below, the (\S) phrase creates a token whenever regexp matches any nonwhitespace character in the string. The second part of the expression, '\1', looks for a second instance of the same character immediately following the first:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

[mat tok ext] = regexp(poestr, '(\S)\1', 'match', ...
    'tokens', 'tokenExtents');
mat
mat =
    'dd'    'pp'    'dd'    'pp'
```

The tokens returned in cell array tok are:

```
'd', 'p', 'd', 'p'
```

Starting and ending indices for each token in the input string poestr are:

```
11 11,  26 26,  35 35,  57 57
```

Using Tokens — Example 1

Here is an example of how tokens are assigned values. Suppose that you are going to search the following text:

```
andy ted bob jim andrew andy ted mark
```

You choose to search the above text with the following search pattern:

```
and(y|rew)|(t)e(d)
```

This pattern has three parenthetical expressions that generate tokens. When you finally perform the search, the following tokens are generated for each match.

Match	Token 1	Token 2
andy	y	
ted	t	d
andrew	rew	
andy	y	
ted	t	d

Only the highest level parentheses are used. For example, if the search pattern and(y|rew) finds the text andrew, token 1 is assigned the value rew. However, if the search pattern (and(y|rew)) is used, token 1 is assigned the value andrew.

Using Tokens — Example 2

Use (expr) and \N to capture pairs of matching HTML tags (e.g., <a> and) and the text between them. The expression used for this example is

```
expr = '<(\w+).*?>.*?</\1>';
```

The first part of the expression, '<(\w+)', matches an opening bracket (<) followed by one or more alphabetic, numeric, or underscore characters. The enclosing parentheses capture token characters following the opening bracket.

The second part of the expression, '.*?>.*?', matches the remainder of this HTML tag (characters up to the >), and any characters that may precede the next opening bracket.

The last part, '</\1>', matches all characters in the ending HTML tag. This tag is composed of the sequence </tag>, where tag is whatever characters were captured as a token.

```
hstr = '<!comment><a name="752507"></a><b>Default</b><br>';
expr = '<(\w+).*?>.*?</\1>';

[mat tok] = regexp(hstr, expr, 'match', 'tokens');
mat{:}
ans =
    <a name="752507"></a>
ans =
    <b>Default</b>

tok{:}
ans =
    'a'
ans =
    'b'
```

Tokens That Are Not Matched

For those tokens specified in the regular expression that have no match in the string being evaluated, regexp and regexpi return an empty string ('') as the token output, and an extent that marks the position in the string where the token was expected.

The example shown here executes regexp on the path string str returned from the MATLAB tempdir function. The regular expression expr includes six token specifiers, one for each piece of the path string. The third specifier

[a-z]+ has no match in the string because this part of the path, `Profiles`, begins with an uppercase letter:

```
str = tempdir
str =
   C:\WINNT\Profiles\bpascal\LOCALS~1\Temp\

expr = ['([A-Z]:)\\(WINNT)\\([a-z]+)?.*\\' ...
        '([a-z]+)\\([A-Z]+~\d)\\(Temp)\\'];

[tok ext] = regexp(str, expr, 'tokens', 'tokenExtents');
```

When a token is not found in a string, MATLAB still returns a token string and token extent. The returned token string is an empty character string (''). The first number of the extent is the string index that marks where the token was expected, and the second number of the extent is equal to one less than the first.

In the case of this example, the empty token is the third specified in the expression, so the third token string returned is empty:

```
tok{:}
ans =
    'C:'    'WINNT'      ''     'bpascal'    'LOCALS~1'    'Temp'
```

The third token extent returned in the variable `ext` has the starting index set to 10, which is where the nonmatching substring, `Profiles`, begins in the string. The ending extent index is set to one less than the starting index, or 9:

```
ext{:}
ans =
     1     2
     4     8
    10     9
    19    25
    27    34
    36    39
```

Using Tokens in a Replacement String

When using tokens in a replacement string, reference them using $1, $2, etc. instead of \1, \2, etc. This example captures two tokens and reverses their order. The first, $1, is 'Norma Jean' and the second, $2, is 'Baker'. Note that regexprep returns the modified string, not a vector of starting indices.

```
regexprep('Norma Jean Baker', '(\w+\s\w+)\s(\w+)', '$2, $1')
ans =
    Baker, Norma Jean
```

Named Capture

If you use a lot of tokens in your expressions, it may be helpful to assign them names rather than having to keep track of which token number is assigned to which token. Use the following operator to assign a name to a token that finds a match.

Operator	Usage
(?<name>expr)	Capture in a token all characters matched by the expression within the parentheses. Assign a name to the token.
\k<name>	Match the token referred to by name.
$<name>	Insert the match for named token in a replacement string. Used only with the regexprep function.
(?(name)s1|s2)	If named token is found, then match s1; otherwise, match s2

When referencing a named token within the expression, use the syntax \k<name> instead of the numeric \1, \2, etc.:

```
poestr = ['While I nodded, nearly napping, ' ...
          'suddenly there came a tapping,'];

regexp(poestr, '(?<anychar>.)\k<anychar>', 'match')
ans =
    'dd'    'pp'    'dd'    'pp'
```

Labeling Your Output

Named tokens can also be useful in labeling the output from the MATLAB regular expression functions. This is especially true when you are processing numerous strings.

This example parses different pieces of street addresses from several strings. A short name is assigned to each token in the expression string:

```
str1 = '134 Main Street, Boulder, CO, 14923';
str2 = '26 Walnut Road, Topeka, KA, 25384';
str3 = '847 Industrial Drive, Elizabeth, NJ, 73548';

p1 = '(?<adrs>\d+\s\S+\s(Road|Street|Avenue|Drive))';
p2 = '(?<city>[A-Z][a-z]+)';
p3 = '(?<state>[A-Z]{2})';
p4 = '(?<zip>\d{5})';

expr = [p1 ', ' p2 ', ' p3 ', ' p4];
```

As the following results demonstrate, you can make your output easier to work with by using named tokens:

```
loc1 = regexp(str1, expr, 'names')
loc1 =
     adrs: '134 Main Street'
     city: 'Boulder'
    state: 'CO'
      zip: '14923'

loc2 = regexp(str2, expr, 'names')
loc2 =
     adrs: '26 Walnut Road'
     city: 'Topeka'
    state: 'KA'
      zip: '25384'

loc3 = regexp(str3, expr, 'names')
loc3 =
     adrs: '847 Industrial Drive'
     city: 'Elizabeth'
```

```
    state: 'NJ'
      zip: '73548'
```

Conditional Expressions

With conditional expressions, you can tell MATLAB to match an expression only if a certain condition is true. A conditional expression is similar to an if-then or an if-then-else clause in programming. MATLAB first tests the state of a given condition, and the outcome of this tests determines what, if anything, is to be matched next. The following table shows the two conditional syntaxes you can use with MATLAB.

Operator	Usage
(?(cond)expr)	If condition cond is true, then match expression expr
(?(cond)expr$_1$|expr$_2$)	If condition cond is true, then match expression expr$_1$. Otherwise match expression expr$_2$

The first entry in this table is the same as an if-then statement. MATLAB tests the state of condition cond and then matches expression expr only if the condition was found to be true. In the form of an if-then statement, it would look like this:

```
if cond then expr
```

The second entry in the table is the same as an if-then-else statement. If the condition is true, MATLAB matches expr$_1$; if false, it matches expr$_2$ instead. This syntax is equivalent to the following programming statement:

```
if cond then expr1 else expr2
```

The condition cond in either of these syntaxes can be any one of the following:

• A specific token, identified by either number or name, is located in the input string. See "Conditions Based on Tokens" on page 2-110, below.

• A lookaround operation results in a match. See "Conditions Based on a Lookaround Match" on page 2-111, below.

• A dynamic expression of the form (?@cmd) returns a nonzero numeric value. See "Conditions Based on Return Values" on page 2-111, below.

Conditions Based on Tokens

In a conditional expression, MATLAB matches the expression only if the condition associated with it is met. If the condition is based on a token, then the condition is met if MATLAB matches more than one character for the token in the input string.

To specify a token in a condition, use either the token number or, for tokens that you have assigned a name to, its name. Token numbers are determined by the order in which they appear in an expression. For example, if you specify three tokens in an expression (that is, if you enclose three parts of the expression in parentheses), then you would refer to these tokens in a condition statement as 1, 2, and 3.

The following example uses the conditional statement `(?(1)her|his)` to match the string regardless of the gender used. You could translate this into the phrase, "**if** token 1 is found (i.e., `Mr` is followed by the letter `s`), **then** match `her`, **else** match `his`:

```
expr = 'Mr(s?)\..*?(?(1)her|his) son';

[mat tok] = regexp('Mr. Clark went to see his son', ...
   expr, 'match', 'tokens')
mat =
    'Mr. Clark went to see his son'
tok =
    {1x2 cell}

tok{:}
ans =
    ''      'his'
```

In the second part of the example, the token `s` is found and MATLAB matches the word `her`:

```
[mat tok] = regexp('Mrs. Clark went to see her son', ...
expr, 'match', 'tokens')
mat =
    'Mrs. Clark went to see her son'
tok =
    {1x2 cell}
```

```
tok{:}
ans =
    's'    'her'
```

Note When referring to a token within a condition, use just the number of the token. For example, refer to token 2 by using the number 2 alone, and not \2 or $2.

Conditions Based on a Lookaround Match

Lookaround statements look for text that either precedes or follows an expression. If this lookaround text is located, then MATLAB proceeds to match the expression. You can also use lookarounds in conditional statements. In this case, if the lookaround text is located, then MATLAB considers the condition to be met and matches the associated expression. If the condition is not met, then MATLAB matches the else part of the expression.

Conditions Based on Return Values

MATLAB supports different types of dynamic expressions. One type of dynamic expression, having the form (?@cmd), enables you to execute a MATLAB command (shown here as cmd) while matching an expression. You can use this type of dynamic expression in a conditional statement if the command in the expression returns a numeric value. The condition is considered to be met if the return value is nonzero.

Dynamic Regular Expressions

In a dynamic expression, you can make the pattern that you want regexp to match dependent on the content of the input string. In this way, you can more closely match varying input patterns in the string being parsed. You can also use dynamic expressions in replacement strings for use with the regexprep function. This gives you the ability to adapt the replacement text to the parsed input.

You can include any number of dynamic expressions in the match_expr or replace_expr arguments of these commands:

```
regexp(string, match_expr)
regexpi(string, match_expr)
regexprep(string, match_expr, replace_expr)
```

MATLAB supports three types of dynamic operators for use in a match expression. See "Dynamic Operators for the Match Expression" on page 2-113 for more information.

Operator	Usage
(??expr)	Parse expr as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called regexprep inside of a regexp match expression.
(?@cmd)	Execute the MATLAB command cmd, discarding any output that may be returned. This is often used for diagnosing a regular expression.
(??@cmd)	Execute the MATLAB command cmd, and include the string returned by cmd in the match expression. This is a combination of the two dynamic syntaxes shown above: (??expr) and (?@cmd).

MATLAB supports one type of dynamic expression for use in the replacement expression of a regexprep command. See "Dynamic Operators for the Replacement Expression" on page 2-118 for more information.

Operator	Usage
${cmd}	Execute the MATLAB command cmd, and include the string returned by cmd in the replacement expression.

Example of a Dynamic Expression

As an example of a dynamic expression, the following regexprep command correctly replaces the term internationalization with its abbreviated form, i18n. However, to use it on a different term such as globalization, you have to use a different replacement expression:

```
match_expr = '(^\w)(\w*)(\w$)';
```

```
replace_expr1 = '$118$3';
regexprep('internationalization', match_expr, replace_expr1)
ans =
    i18n

replace_expr2 = '$111$3';
regexprep('globalization', match_expr, replace_expr2)
ans =
    g11n
```

Using a dynamic expression `${num2str(length($2))}` enables you to base the replacement expression on the input string so that you do not have to change the expression each time. This example uses the dynamic syntax `${cmd}` from the second table shown above:

```
match_expr = '(^\w)(\w*)(\w$)';
replace_expr = '$1${num2str(length($2))}$3';

regexprep('internationalization', match_expr, replace_expr)
ans =
    i18n

regexprep('globalization', match_expr, replace_expr)
ans =
    g11n
```

Dynamic Operators for the Match Expression

There are three types of dynamic expressions you can use when composing a match expression:

- "Dynamic Expressions That Modify the Match Expression — (??expr)" on page 2-114

- "Dynamic Commands That Modify the Match Expression — (??@cmd)" on page 2-115

- "Dynamic Commands That Serve a Functional Purpose — (?@cmd)" on page 2-116

The first two of these actually modify the match expression itself so that it can be made specific to changes in the contents of the input string. When

MATLAB evaluates one of these dynamic statements, the results of that evaluation are included in the same location within the overall match expression.

The third operator listed here does not modify the overall expression, but instead enables you to run MATLAB commands during the parsing of a regular expression. This functionality can be useful in diagnosing your regular expressions.

Dynamic Expressions That Modify the Match Expression — (??expr).
The (??expr) operator parses expression expr, and inserts the results back into the match expression. MATLAB then evaluates the modified match expression.

Here is an example of the type of expression that you can use with this operator:

```
str = {'5XXXXX', '8XXXXXXXX', '1X'};
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
```

The purpose of this particular command is to locate a series of X characters in each of the strings stored in the input cell array. Note however that the number of Xs varies in each string. If the count did not vary, you could use the expression X{n} to indicate that you want to match n of these characters. But, a constant value of n does not work in this case.

The solution used here is to capture the leading count number (e.g., the 5 in the first string of the cell array) in a token, and then to use that count in a dynamic expression. The dynamic expression in this example is (??X{$1}), where $1 is the value captured by the token \d+. The operator {$1} makes a quantifier of that token value. Because the expression is dynamic, the same pattern works on all three of the input strings in the cell array. With the first input string, regexp looks for five X characters; with the second, it looks for eight, and with the third, it looks for just one:

```
regexp(str, '^(\d+)(??X{$1})$', 'match', 'once')
ans =
    '5XXXXX'    '8XXXXXXXX'    '1X'
```

Dynamic Commands That Modify the Match Expression — (??@cmd).
MATLAB uses the (??@function) operator to include the results of a
MATLAB command in the match expression. This command must return a
string that can be used within the match expression.

The regexp command below uses the dynamic expression (??@flilplr($1))
to locate a palindrome string, "Never Odd or Even", that has been embedded
into a larger string:

```
regexp(pstr, '(.{3,}).?(??@fliplr($1))', 'match')
```

The dynamic expression reverses the order of the letters that make up the
string, and then attempts to match as much of the reversed-order string as
possible. This requires a dynamic expression because the value for $1 relies
on the value of the token (.{3,}):

```
% Put the string in lowercase.
str = lower(...
  'Find the palindrome Never Odd or Even in this string');

% Remove all nonword characters.
str = regexprep(str, '\W*', '')
str =
   findthepalindromeneveroddoreveninthisstring

% Now locate the palindrome within the string.
palstr = regexp(str, '(.{3,}).?(??@fliplr($1))', 'match')
str =
   'neveroddoreven'
```

Dynamic expressions in MATLAB have access to the currently active
workspace. This means that you can change any of the functions or variables
used in a dynamic expression just by changing variables in the workspace.
Repeat the last command of the example above, but this time define the
function to be called within the expression using a function handle stored in
the base workspace:

```
fun = @fliplr;

palstr = regexp(str, '(.{3,}).?(??@fun($1))', 'match')
palstr =
```

```
'neveroddoreven'
```

Dynamic Commands That Serve a Functional Purpose — (?@cmd). The
(?@cmd) operator specifies a MATLAB command that `regexp` or `regexprep`
is to run while parsing the overall match expression. Unlike the other
dynamic expressions in MATLAB, this operator does not alter the contents
of the expression it is used in. Instead, you can use this functionality to get
MATLAB to report just what steps it is taking as it parses the contents of one
of your regular expressions.

The following example parses a word for zero or more characters followed by
two identical characters followed again by zero or more characters:

```
regexp('mississippi', '\w*(\w)\1\w*', 'match')
ans =
    'mississippi'
```

To track the exact steps that MATLAB takes in determining the match, the
example inserts a short script (?@disp($1)) in the expression to display
the characters that finally constitute the match. Because the example uses
greedy quantifiers, MATLAB attempts to match as much of the string as
possible. So, even though MATLAB finds a match toward the beginning of the
string, it continues to look for more matches until it arrives at the very end of
the string. From there, it backs up through the letters i then p and the next
p, stopping at that point because the match is finally satisfied:

```
regexp('mississippi', '\w*(\w)(?@disp($1))\1\w*');
i
p
p
```

Now try the same example again, this time making the first quantifier lazy
(*?). Again, MATLAB makes the same match:

```
regexp('mississippi', '\w*?(\w)\1\w*', 'match')
ans =
    'mississippi'
```

But by inserting a dynamic script, you can see that this time, MATLAB has
matched the string quite differently. In this case, MATLAB uses the very first
match it can find, and does not even consider the rest of the string:

```
regexp('mississippi', '\w*?(\w)(?@disp($1))\1\w*';)
m
i
s
```

To demonstrate how versatile this type of dynamic expression can be, consider the next example that progressively assembles a cell array as MATLAB iteratively parses the input string. The (?!) operator found at the end of the expression is actually an empty lookahead operator, and forces a failure at each iteration. This forced failure is necessary if you want to trace the steps that MATLAB is taking to resolve the expression.

MATLAB makes a number of passes through the input string, each time trying another combination of letters to see if a fit better than last match can be found. On any passes in which no matches are found, the test results in an empty string. The dynamic script (?@if(~isempty($&))) serves to omit these strings from the matches cell array:

```
matches = {};
expr = ['(Euler\s)?(Cauchy\s)?(Boole)?(?@if(~isempty($&)),' ...
    'matches{end+1}=$&;end)(?!)'];

regexp('Euler Cauchy Boole', expr);

matches
matches =
    'Euler Cauchy Boole'    'Euler Cauchy '    'Euler '
'Cauchy Boole'    'Cauchy '    'Boole'
```

The operators $& (or the equivalent $0), $`, and $' refer to that part of the input string that is currently a match, all characters that precede the current match, and all characters to follow the current match, respectively. These operators are sometimes useful when working with dynamic expressions, particularly those that employ the (?@cmd) operator.

This example parses the input string looking for the letter g. At each iteration through the string, regexp compares the current character with g, and not finding it, advances to the next character. The example tracks the progress of scan through the string by marking the current location being parsed with a ^ character.

2-117

(The $\` and $· operators capture that part of the string that precedes and follows the current parsing location. You need two single-quotation marks ($'') to express the sequence $· when it appears within a string.)

```
str = 'abcdefghij';
expr = '(?@disp(sprintf(''starting match: [%s^%s]'',$`,$'')))g';

regexp(str, expr, 'once');
starting match: [^abcdefghij]
starting match: [a^bcdefghij]
starting match: [ab^cdefghij]
starting match: [abc^defghij]
starting match: [abcd^efghij]
starting match: [abcde^fghij]
starting match: [abcdef^ghij]
```

Dynamic Operators for the Replacement Expression

The three types of dynamic expressions discussed above can be used only in the match expression (second input) argument of the regular expression functions. MATLAB provides one more type of dynamic expression; this one is for use in a replacement string (third input) argument of the regexprep function.

Dynamic Commands That Modify the Replacement Expression — ${cmd}. The ${cmd} operator modifies the contents of a regular expression replacement string, making this string adaptable to parameters in the input string that might vary from one use to the next. As with the other dynamic expressions used in MATLAB, you can include any number of these expressions within the overall replacement expression.

In the regexprep call shown here, the replacement string is '${convert($1,$2)}'. In this case, the entire replacement string is a dynamic expression:

```
regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

The dynamic expression tells MATLAB to execute a function named convert using the two tokens (\d+\.?\d*) and (\w+), derived from the string being

matched, as input arguments in the call to convert. The replacement string requires a dynamic expression because the values of $1 and $2 are generated at runtime.

The following example defines the file named convert that converts measurements from imperial units to metric. To convert values from the string being parsed, regexprep calls the convert function, passing in values for the quantity to be converted and name of the imperial unit:

```
function valout  = convert(valin, units)
switch(units)
    case 'inches'
        fun = @(in)in .* 2.54;    uout = 'centimeters';
    case 'miles'
        fun = @(mi)mi .* 1.6093;  uout = 'kilometers';
    case 'pounds'
        fun = @(lb)lb .* 0.4536;  uout = 'kilograms';
    case 'pints'
        fun = @(pt)pt .* 0.4731;  uout = 'litres';
    case 'ounces'
        fun = @(oz)oz .* 28.35;   uout = 'grams';
end
val = fun(str2num(valin));
valout = [num2str(val) ' ' uout];


regexprep('This highway is 125 miles long', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
    This highway is 201.1625 kilometers long


regexprep('This pitcher holds 2.5 pints of water', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
ans =
    This pitcher holds 1.1828 litres of water


regexprep('This stone weighs about 10 pounds', ...
          '(\d+\.?\d*)\W(\w+)', '${convert($1,$2)}')
```

```
ans =
    This stone weighs about 4.536 kilograms
```

As with the (??@) operator discussed in an earlier section, the ${ } operator
has access to variables in the currently active workspace. The following
regexprep command uses the array A defined in the base workspace:

```
A = magic(3)
A =
    8    1    6
    3    5    7
    4    9    2

regexprep('The columns of matrix _nam are _val', ...
          {'_nam', '_val'}, ...
          {'A', '${sprintf(''%d%d%d '', A)}'})
ans =
The columns of matrix A are 834 159 672
```

String Replacement

The regexprep function enables you to replace a string that is identified
by a regular expression with another string. The following syntax replaces
all occurrences of the regular expression expr in string str with the string
repstr. The new string is returned in s. If no matches are found, return
string s is the same as input string str.

```
s = regexprep('str', 'expr', 'repstr')
```

The replacement string can include any ordinary characters and also any of
the operators shown in the following table:

Operator	Usage
Operators from the "Character Representation" on page 2-88 table	The character represented by the operator sequence
$`	That part of the input string that precedes the current match
$& or $0	That part of the input string that is currently a match

Operator	Usage
`$'`	That part of the input string that follows the current match. In MATLAB, use `$''` to represent the character sequence `$'`
`$N`	The string represented by the token identified by the number `N`
`$<name>`	The string represented by the token identified by `name`
`${cmd}`	The string returned when MATLAB executes the command `cmd`

You can capture parts of the input string as tokens and then reuse them in the replacement string. Specify the parts of the string to capture using the token capture operator `(...)`. Specify the tokens to use in the replacement string using the operators `$1`, `$2`, `$N` to reference the first, second, and `N`th tokens captured. (See the section on "Tokens" on page 2-102 and the example "Using Tokens in a Replacement String" on page 2-107 in this documentation for information on using tokens.)

Note When referring to a token within a replacement string, use the number of the token preceded by a dollar sign. For example, refer to token `2` by using `$2`, and not `2` or `\2`.

The following example uses both the `${cmd}` and `$N` operators in the replacement strings of nested `regexprep` commands to capitalize the first letter of each sentence. The inner `regexprep` looks for the start of the entire string and capitalizes the single instance; the outer `regexprep` looks for the first letter following a period and capitalizes the two instances:

```
s1 = 'here are a few sentences.';
s2 = 'none are capitalized.';
s3 = 'let''s change that.';
str = [s1 ' ' s2 ' ' s3]

regexprep(regexprep(str, '(^.)', '${upper($1)}'), ...
```

```
        '(?<=\.\s*)([a-z])','${upper($1)}')

    ans =
    Here are a few sentences. None are capitalized. Let's change that.
```

Make `regexprep` more specific to your needs by specifying any of a number of options with the command. See the `regexprep` reference page for more information on these options.

Handling Multiple Strings

You can use any of the MATLAB regular expression functions with cell arrays of strings as well as with single strings. Any or all of the input parameters (the string, expression, or replacement string) can be a cell array of strings. The `regexp` function requires that the string and expression arrays have the same number of elements. The `regexprep` function requires that the expression and replacement arrays have the same number of elements. (The cell arrays do not have to have the same shape.)

Whenever either input argument in a call to `regexp`, or the first input argument in a call to `regexprep` function is a cell array, all output values are cell arrays of the same size.

This section covers the following topics:

- "Finding a Single Pattern in Multiple Strings" on page 2-122
- "Finding Multiple Patterns in Multiple Strings" on page 2-123
- "Replacing Multiple Strings" on page 2-124

Finding a Single Pattern in Multiple Strings

The example shown here uses the `regexp` function on a cell array of strings `cstr`. It searches each string of the cell array for consecutive matching letters (e.g., `'oo'`). The function returns a cell array of the same size as the input array. Each row of the return array contains the indices for which there was a match against the input cell array.

Here is the input cell array:

```
    cstr = {                                        ...
```

```
'Whose woods these are I think I know.' ; ...
'His house is in the village though;'   ; ...
'He will not see me stopping here'      ; ...
'To watch his woods fill up with snow.'};
```

Find consecutive matching letters by capturing a letter as a token (.) and then repeating that letter as a token reference, \1:

```
idx = regexp(cstr, '(.)\1');

whos idx
  Name       Size                     Bytes  Class

   idx        4x1                        296  cell array

idx{:}
ans =                     % 'Whose woods these are I think I know.'
    8                     %           |8

ans =                     % 'His house is in the village though;'
   23                     %                             |23

ans =                     % 'He will not see me stopping here'
    6    14    23         %         |6      |14      |23

ans =                     % 'To watch his woods fill up with snow.'
   15    22               %                       |15    |22
```

To return substrings instead of indices, use the 'match' parameter:

```
mat = regexp(cstr, '(.)\1', 'match');
mat{3}
ans =
   'll'    'ee'    'pp'
```

Finding Multiple Patterns in Multiple Strings

This example uses a cell array of strings in both the input string and the expression. The two cell arrays are of different shapes: cstr is 4-by-1 while expr is 1-by-4. The command is valid as long as they both have the same number of cells.

Find uppercase or lowercase `'i'` followed by a white-space character in `str{1}`, the sequence `'hou'` in `str{2}`, two consecutive matching letters in `str{3}`, and words beginning with `'w'` followed by a vowel in `str{4}`.

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
idx = regexpi(cstr, expr);

idx{:}
ans =                      % 'Whose woods these are I think I know.'
    23    31               %                                |23      |31

ans =                      % 'His house is in the village though;'
     5    30               %      |5                             |30

ans =                      % 'He will not see me stopping here'
     6    14    23         %        |6        |14        |23

ans =                      % 'To watch his woods fill up with snow.'
     4    14    28         %     |4          |14               |28
```

Note that the returned cell array has the dimensions of the input string, `cstr`. The dimensions of the return value are always derived from the input string, whenever the input string is a cell array. If the input string is not a cell array, then it is the dimensions of the expression that determine the shape of the return array.

Replacing Multiple Strings

When replacing multiple strings with `regexprep`, use a single replacement string if the expression consists of a single string. This example uses a common replacement value (`'--'`) for all matches found in the multiple string input `cstr`. The function returns a cell array of strings having the same dimensions as the input cell array:

```
s = regexprep(cstr, '(.)\1', '--', 'ignorecase')
s =
    'Whose w--ds these are I think I know.'
    'His house is in the vi--age though;'
    'He wi-- not s-- me sto--ing here'
    'To watch his w--ds fi-- up with snow.'
```

You can use multiple replacement strings if the expression consists of multiple strings. In this example, the input string and replacement string are both 4-by-1 cell arrays, and the expression is a 1-by-4 cell array. As long as the expression and replacement arrays contain the same number of elements, the statement is valid. The dimensions of the return value match the dimensions of the input string:

```
expr = {'i\s', 'hou', '(.)\1', '\<w[aeiou]'};
repl = {'-1-'; '-2-'; '-3-'; '-4-'};

s = regexprep(cstr, expr, repl, 'ignorecase')
s =
    'Whose w-3-ds these are -1-think -1-know.'
    'His -2-se is in the vi-3-age t-2-gh;'
    'He -4--3- not s-3- me sto-3-ing here'
    'To -4-tch his w-3-ds fi-3- up -4-th snow.'
```

Function, Mode Options, Operator, Return Value Summaries

- "Function Summary" on page 2-125

- "Mode Options Summary" on page 2-126

- "Operator Summary" on page 2-127

- "Return Value Summary" on page 2-133

Function Summary

MATLAB Regular Expression Functions

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.

MATLAB Regular Expression Functions (Continued)

Function	Description
regexprep	Replace string using regular expression.
regexptranslate	Translate string into regular expression.

Mode Options Summary

Mode Keyword	Flag	Description
ignorecase	(?i)	Do not consider letter case when matching patterns to a string. (The default for regexpi)
matchcase	(?-i)	Letter case must match when matching patterns to a string. (The default for regexp)
dotall	(?s)	Match dot ('.') in the pattern string with any character. (The default)
dotexceptnewline	(?-s)	Match dot in the pattern with any character that is not a newline.
lineanchors	(?m)	Match the ^ and $ metacharacters at the beginning and end of a line.
stringanchors	(?-m)	Match the ^ and $ metacharacters at the beginning and end of a string (the default).
freespacing	(?x)	Ignore spaces and comments when parsing the string. (You must use '\ ' and '\#' to match space and # characters.)
literalspacing	(?-x)	Parse space characters and comments (the # character and any text to the right of it) in the same way as any other characters in the string. (The default)

Operator Summary

Character Types

Operator	Usage
.	Any single character, including white space
[$c_1 c_2 c_3$]	Any character contained within the brackets: c_1 or c_2 or c_3
[^$c_1 c_2 c_3$]	Any character not contained within the brackets: anything but c_1 or c_2 or c_3
[c_1-c_2]	Any character in the range of c_1 through c_2
\s	Any white-space character; equivalent to [\f\n\r\t\v]
\S	Any nonwhitespace character; equivalent to [^ \f\n\r\t\v]
\w	Any alphabetic, numeric, or underscore character (For English character sets, this is equivalent to [a-zA-Z_0-9].)
\W	Any character that is not alphabetic, numeric, or underscore (For English character sets, this is equivalent to [^a-zA-Z_0-9].)
\d	Any numeric digit; equivalent to [0-9]
\D	Any nondigit character; equivalent to [^0-9]
\oN or \o{N}	Character of octal value N
\xN or \x{N}	Character of hexadecimal value N

Character Representation

Operator	Usage
\\	Backslash
\a	Alarm (beep)
\b	Backspace

Character Representation (Continued)

Operator	Usage
\f	Form feed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\char	If a character has special meaning in a regular expression, precede it with backslash (\) to match it literally.

Grouping Operators

Operator	Usage
(expr)	Group regular expressions and capture tokens.
(?:expr)	Group regular expressions, but do not capture tokens.
(?>expr)	Group atomically.
$expr_1|expr_2$	Match expression $expr_1$ or expression $expr_2$.

Nonmatching Operators

Operator	Usage
(?#comment)	Insert a comment into the expression. Comments are ignored in matching.

Positional Operators

Operator	Usage
^expr	Match expr if it occurs at the beginning of the input string.
expr$	Match expr if it occurs at the end of the input string.

Positional Operators (Continued)

Operator	Usage
\<expr	Match expr when it occurs at the beginning of a word.
expr\>	Match expr when it occurs at the end of a word.
\<expr\>	Match expr when it represents the entire word.

Lookaround Operators

Operator	Usage
(?=expr)	Look ahead from current position and test if expr is found.
(?!expr)	Look ahead from current position and test if expr is not found
(?<=expr)	Look behind from current position and test if expr is found.
(?<!expr)	Look behind from current position and test if expr is not found.

Quantifiers

Operator	Usage
expr{m,n}	Match expr when it occurs at least m times but no more than n times consecutively.
expr{m,}	Match expr when it occurs at least m times consecutively.
expr{n}	Match expr when it occurs exactly n times consecutively. Equivalent to {n,n}.
expr?	Match expr when it occurs 0 times or 1 time. Equivalent to {0,1}.
expr*	Match expr when it occurs 0 or more times consecutively. Equivalent to {0,}.

Quantifiers (Continued)

Operator	Usage
expr+	Match expr when it occurs 1 or more times consecutively. Equivalent to {1,}.
q_expr	Match as much of the quantified expression as possible, where q_expr represents any of the expressions shown in the first six rows of this table.
q_expr+	Match as much of the quantified expression as possible, but do not rescan any portions of the string if the initial match fails.
q_expr?	Match only as much of the quantified expression as necessary.

Ordinal Token Operators

Operator	Usage
(expr)	Capture in a token all characters matched by the expression within the parentheses.
\N	Match the Nth token generated by this command. That is, use \1 to match the first token, \2 to match the second, and so on.
$N	Insert the match for the Nth token in the replacement string. If N is equal to zero, then insert the entire match in the replacement string. (Used only by the regexprep function.)
(?(N)s1|s2)	If N^{th} token is found, then match s1; otherwise, match s2.

Named Token Operators

Operator	Usage
`(?<name>expr)`	Capture in a token all characters matched by the expression within the parentheses. Assign a `name` value to the token.
`\k<name>`	Match the token referred to by `name`.
`$<name>`	Insert the match for named token in a replacement string. (Used only with the `regexprep` function.)
`(?(name)s1|s2)`	If named token is found, then match `s1`; otherwise, match `s2`.

Conditional Expression Operators

Operator	Usage
`(?(cond)expr)`	If condition `cond` is `true`, then match expression `expr`.
`(?(cond)expr₁|expr₂)`	If condition `cond` is `true`, then match expression $expr_1$. Otherwise match expression $expr_2$.

Dynamic Expression Operators

Operator	Usage
`(??expr)`	Parse `expr` as a separate regular expression, and include the resulting string in the match expression. This gives you the same results as if you called `regexprep` inside of a `regexp` match expression.
`(??@cmd)`	Execute the MATLAB command represented by `cmd`, and include the string returned by the command in the match expression. This is a combination of the two dynamic syntaxes shown previously: `(??expr)` and `(?@cmd)`.

Dynamic Expression Operators (Continued)

Operator	Usage
(?@cmd)	Execute the MATLAB command represented by cmd and discard any output the command returns. (Helpful for diagnosing regular expressions).
${cmd}	Execute the MATLAB command represented by cmd, and include the string returned by the command in the replacement expression.

Replacement String Operators

Operator	Usage
Operators from "Character Representation" on page 2-88 table	The character represented by the operator sequence
$'	That part of the input string that precedes the current match
$& or $0	That part of the input string that is currently a match
$'	That part of the input string that follows the current match (In MATLAB, use $'' to represent the character sequence $'.)
$N	The string represented by the token identified by name
$<name>	The string represented by the token identified by name
${cmd}	The string returned when MATLAB executes the command cmd

Return Value Summary

Qualifier	Description	Default Order
start	Row vector containing the starting index of each substring of str that matches expr	1
end	Row vector containing the ending index of each substring of str that matches expr	2
tokenExtents	Cell array containing the starting and ending indices of each substring of str that matches a token in expr (This is a double array when used with 'once'.)	3
match	Cell array containing the text of each substring of str that matches expr (This is a string when used with 'once'.)	4
tokens	Cell array of cell arrays of strings containing the text of each token captured by regexp (This is a cell array of strings when used with 'once'.)	5
names	Structure array containing the name and text of each *named* token captured by regexp (If there are no named tokens in expr, regexp returns a structure array with no fields.) Field names of the returned structure are set to the token names, and field values are the text of those tokens. Named tokens are generated by the expression (?<tokenname>).	6
split	Cell array containing those parts of the input string that are delimited by substrings returned when using the regexp 'match' option	7

Symbol Reference

This section does not include symbols used in arithmetic, relational, and logical operations. For a description of these symbols, see the top of the Alphabetical List of functions in the MATLAB Help browser.

Asterisk — *

An asterisk in a filename specification is used as a wildcard specifier, as described below.

Filename Wildcard

Wildcards are generally used in file operations that act on multiple files or folders. They usually appear in the string containing the file or folder specification. MATLAB matches all characters in the name exactly except for the wildcard character `*`, which can match any one or more characters.

To locate all files with names that start with `'january_'` and have a `mat` file extension, use

```
dir('january_*.mat')
```

You can also use wildcards with the `who` and `whos` functions. To get information on all variables with names starting with `'image'` and ending with `'Offset'`, use

```
whos image*Offset
```

At — @

The `@` sign signifies either a function handle constructor or a folder that supports a MATLAB class.

Function Handle Constructor

The `@` operator forms a handle to either the named function that follows the `@` sign, or to the anonymous function that follows the `@` sign.

Function Handles in General. Function handles are commonly used in passing functions as arguments to other functions. Construct a function handle by preceding the function name with an `@` sign:

```
fhandle = @myfun
```

You can read more about function handles in "Function Handles" on page 1-127.

2-135

Handles to Anonymous Functions. Anonymous functions give you a quick means of creating simple functions without having to create your function in a file each time. You can construct an anonymous function and a handle to that function using the syntax

```
fhandle = @(arglist) body
```

where `body` defines the body of the function and `arglist` is the list of arguments you can pass to the function.

See "Anonymous Functions" on page 4-3 for more information.

Class Folder Designator

An `@` sign can indicate the name of a class folder, such as

```
\@myclass\get.m
```

See the documentation on "Options for Class Folders" for more information.

Colon — :

The colon operator generates a sequence of numbers that you can use in creating or indexing into arrays. See"Generating a Numeric Sequence" for more information on using the colon operator.

Numeric Sequence Range

Generate a sequential series of regularly spaced numbers from `first` to `last` using the syntax `first:last`. For an incremental sequence from 6 to 17, use

```
N = 6:17
```

Numeric Sequence Step

Generate a sequential series of numbers, each number separated by a `step` value, using the syntax `first:step:last`. For a sequence from 2 through 38, stepping by 4 between each entry, use

```
N = 2:4:38
```

Indexing Range Specifier

Index into multiple rows or columns of a matrix using the colon operator to specify a range of indices:

```
B = A(7, 1:5);          % Read columns 1-5 of row 7.
B = A(4:2:8, 1:5);      % Read columns 1-5 of rows 4, 6, and 8.
B = A(:, 1:5);          % Read columns 1-5 of all rows.
```

Conversion to Column Vector

Convert a matrix or array to a column vector using the colon operator as a single index:

```
A = rand(3,4);
B = A(:);
```

Preserving Array Shape on Assignment

Using the colon operator on the left side of an assignment statement, you can assign new values to array elements without changing the shape of the array:

```
A = rand(3,4);
A(:) = 1:12;
```

Comma — ,

A comma is used to separate the following types of elements.

Row Element Separator

When constructing an array, use a comma to separate elements that belong in the same row:

```
A = [5.92, 8.13, 3.53]
```

Array Index Separator

When indexing into an array, use a comma to separate the indices into each dimension:

```
X = A(2, 7, 4)
```

Function Input and Output Separator

When calling a function, use a comma to separate output and input arguments:

```
function [data, text] = xlsread(file, sheet, range, mode)
```

Command or Statement Separator

To enter more than one MATLAB command or statement on the same line, separate each command or statement with a comma:

```
for k = 1:10,   sum(A(k)),    end
```

Curly Braces — { }

Use curly braces to construct or get the contents of cell arrays.

Cell Array Constructor

To construct a cell array, enclose all elements of the array in curly braces:

```
C = {[2.6 4.7 3.9], rand(8)*6, 'C. Coolidge'}
```

Cell Array Indexing

Index to a specific cell array element by enclosing all indices in curly braces:

```
A = C{4,7,2}
```

See the documentation on Cell Arrays for more information.

Dot — .

The single dot operator has the following different uses in MATLAB.

Decimal Point

MATLAB uses a period to separate the integral and fractional parts of a number.

Structure Field Definition

Add fields to a MATLAB structure by following the structure name with a dot and then a field name:

```
funds(5,2).bondtype = 'Corporate';
```

See the documentation on "Structures" on page 1-67 for more information.

Object Method Specifier

Specify the properties of an instance of a MATLAB class using the object name followed by a dot, and then the property name:

```
val = asset.current_value
```

See "Defining Your Own Classes" on page 1-172 for more information.

Dot-Dot — ..

Two dots in sequence refer to the parent of the current folder.

Parent Folder

Specify the folder immediately above your current folder using two dots. For example, to go up two levels in the folder tree and down into the test folder, use

```
cd ..\..\test
```

Dot-Dot-Dot (Ellipsis) — ...

A series of three consecutive periods (...) is the line continuation operator in MATLAB. This is often referred to as an *ellipsis*, but it should be noted that the line continuation operator is a three-character operator and is different from the single-character ellipsis represented in ASCII by the hexadecimal number 2026.

Line Continuation

Continue any MATLAB command or expression by placing an ellipsis at the end of the line to be continued:

```
sprintf('The current value of %s is %d', ...
        vname, value)
```

Entering Long Strings. You cannot use an ellipsis within single quotes to continue a string to the next line:

```
string = 'This is not allowed and will generate an ...
    error in MATLAB.'
```

To enter a string that extends beyond a single line, piece together shorter strings using either the concatenation operator ([]) or the sprintf function.

Here are two examples:

```
quote1 = [
 'Tiger, tiger, burning bright in the forests of the night,' ...
 'what immortal hand or eye could frame thy fearful symmetry?'];
quote2 = sprintf('%s%s%s', ...
 'In Xanadu did Kubla Khan a stately pleasure-dome decree,', ...
 'where Alph, the sacred river, ran ', ...
 'through caverns measureless to man down to a sunless sea.');
```

Defining Arrays. MATLAB interprets the ellipsis as a space character. For statements that define arrays or cell arrays within [] or {} operators, a space character separates array elements. For example,

```
not_valid = [1 2 zeros...
(1,3)]
```

is equivalent to

```
not_valid = [1 2 zeros (1,3)]
```

which returns an error. Place the ellipses so that the interpreted statement is valid, such as

```
valid = [1 2 ...
```

```
        zeros(1,3)]
```

Dot-Parentheses — .( )

Use dot-parentheses to specify the name of a dynamic structure field.

Dynamic Structure Fields

Sometimes it is useful to reference structures with field names that can vary. For example, the referenced field might be passed as an argument to a function. Dynamic field names specify a variable name for a structure field.

The variable `fundtype` shown here is a dynamic field name:

```
type = funds(5,2).(fundtype);
```

See "Creating Field Names Dynamically" on page 1-77 for more information.

Exclamation Point — !

The exclamation point precedes operating system commands that you want to execute from within MATLAB.

Shell Escape

The exclamation point initiates a shell escape function. Such a function is to be performed directly by the operating system:

```
!rmdir oldtests
```

See "Shell Escape Functions" on page 2-7 for more information.

Parentheses — ( )

Parentheses are used mostly for indexing into elements of an array or for specifying arguments passed to a called function. Parenthesis also control the order of operations, and can group a vector visually (such as `x = (1:10)`) without calling a concatenation function.

Array Indexing

When parentheses appear to the right of a variable name, they are indices into the array stored in that variable:

```
A(2, 7, 4)
```

Function Input Arguments

When parentheses follow a function name in a function declaration or call, the enclosed list contains input arguments used by the function:

```
function sendmail(to, subject, message, attachments)
```

Percent — %

The percent sign is most commonly used to indicate nonexecutable text within the body of a program. This text is normally used to include comments in your code. Two percent signs, %%, serve as a cell delimiterdescribed in "Evaluating Subsections of Files Using Code Cells". Some functions also interpret the percent sign as a conversion specifier.

Single Line Comments

Precede any one-line comments in your code with a percent sign. MATLAB does not execute anything that follows a percent sign (that is, unless the sign is quoted, '%'):

```
% The purpose of this routine is to compute
% the value of ...
```

See "Help Text" on page 3-13 for more information.

Conversion Specifiers

Some functions, like sscanf and sprintf, precede conversion specifiers with the percent sign:

```
sprintf('%s = %d', name, value)
```

Percent-Brace — %{ %}

The `%{` and `%}` symbols enclose a block of comments that extend beyond one line.

Block Comments

Enclose any multiline comments with percent followed by an opening or closing brace.

```
%{
The purpose of this routine is to compute
the value of ...
%}
```

Note With the exception of whitespace characters, the `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Plus — +

The + sign appears most frequently as an arithmetic operator, but is also used to designate the names of package folders. For more information, see "Scoping Classes with Packages".

Semicolon — ;

The semicolon can be used to construct arrays, suppress output from a MATLAB command, or to separate commands entered on the same line.

Array Row Separator

When used within square brackets to create a new array or concatenate existing arrays, the semicolon creates a new row in the array:

```
A = [5, 8; 3, 4]
A =
     5     8
     3     4
```

Output Suppression

When placed at the end of a command, the semicolon tells MATLAB not to display any output from that command. In this example, MATLAB does not display the resulting 100-by-100 matrix:

```
A = ones(100, 100);
```

Command or Statement Separator

Like the comma operator, you can enter more than one MATLAB command on a line by separating each command with a semicolon. MATLAB suppresses output for those commands terminated with a semicolon, and displays the output for commands terminated with a comma.

In this example, assignments to variables A and C are terminated with a semicolon, and thus do not display. Because the assignment to B is comma-terminated, the output of this one command is displayed:

```
A = 12.5;  B = 42.7,  C = 1.25;
B =
   42.7000
```

Single Quotes — ' '

Single quotes are the constructor symbol for MATLAB character arrays.

Character and String Constructor

MATLAB constructs a character array from all characters enclosed in single quotes. If only one character is in quotes, then MATLAB constructs a 1-by-1 array:

```
S = 'Hello World'
```

See "Characters and Strings" on page 1-35 for more information.

Space Character

The space character serves a purpose similar to the comma in that it can be used to separate row elements in an array constructor, or the values returned by a function.

Row Element Separator

You have the option of using either commas or spaces to delimit the row elements of an array when constructing the array. To create a 1-by-3 array, use

```
A = [5.92 8.13 3.53]
A =
    5.9200    8.1300    3.5300
```

When indexing into an array, you must always use commas to reference each dimension of the array.

Function Output Separator

Spaces are allowed when specifying a list of values to be returned by a function. You can use spaces to separate return values in both function declarations and function calls:

```
function [data text] = xlsread(file, sheet, range, mode)
```

Slash and Backslash — / \

The slash (/) and backslash (\) characters separate the elements of a path or folder string. On Microsoft® Windows®-based systems, both slash and backslash have the same effect. On The Open Group UNIX®-based systems, you must use slash only.

On a Windows system, you can use either backslash or slash:

```
dir([matlabroot '\toolbox\matlab\elmat\shiftdim.m'])
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

On a UNIX system, use only the forward slash:

```
dir([matlabroot '/toolbox/matlab/elmat/shiftdim.m'])
```

Square Brackets — [ ]

Square brackets are used in array construction and concatenation, and also in declaring and capturing values returned by a function.

Array Constructor

To construct a matrix or array, enclose all elements of the array in square brackets:

```
A = [5.7, 9.8, 7.3; 9.2, 4.5, 6.4]
```

Concatenation

To combine two or more arrays into a new array through concatenation, enclose all array elements in square brackets:

```
A = [B, eye(6), diag([0:2:10])]
```

Function Declarations and Calls

When declaring or calling a function that returns more than one output, enclose each return value that you need in square brackets:

```
[data, text] = xlsread(file, sheet, range, mode)
```

Tilde — ~

The tilde character is used as a placeholder for an input or output argument you want to omit from a function call.

Argument Placeholder

To have the fileparts function return its third output value and skip the first two, replace arguments one and two with a tilde character:

```
[~, ~, filenameExt] = fileparts(fileSpec);
```

3

Functions and Scripts

Program Development

In this section...

Overview

As you write a MATLAB function or script, you save it to a file that has a `.m` file extension. There are two types of these files you can write: scripts and functions. This section covers basic program development, describes how to write and call scripts and functions, and shows how to pass different types of data when calling a function. Associated with each step of this process are certain MATLAB tools and utilities that are fully documented in the Desktop Tools and Development Environment documentation.

For more ideas on good programming style, see "Program Development" on page 10-18 in the MATLAB Programming Tips documentation. The Programming Tips section is a compilation of useful pieces of information that can show you alternate and often more efficient ways to accomplish common programming tasks while also expanding your knowledge of MATLAB.

Creating a Program

You can type in your program code using any text editor. This section focuses on using the MATLAB Editor/Debugger for this purpose. The Editor/Debugger is fully documented in Ways to Edit and Debug Files in the Desktop Tools and Development Environment documentation.

The first step in creating a program is to open an editing window. To create a file for a new function, type the word `edit` at the MATLAB command prompt. To edit an existing file, type `edit` followed by the file name:

```
edit drawPlot.m
```

MATLAB opens a new window for entering your program code. As you type in your program, MATLAB keeps track of the line numbers in the left column.

Saving the Program

It is usually a good idea to save your program periodically while you are in the development process. To do this, click **File > Save** in the Editor/Debugger. Enter a file name with a `.m` extension in the **Save file as** dialog box that appears and click **OK**. It is customary and less confusing if you give the file the same name as the first function in the file.

Running the Program

Before trying to run your program, make sure that its file is on the MATLAB path. The MATLAB path defines those folders that you want MATLAB to know about when executing files. The path includes all the folders that contain functions provided with MATLAB. It should also include any folders that you use for your own functions.

Use the `which` function to see if your program is on the path:

```
which drawPlot
   D:\user5\matlab\mywork\drawPlot.m
```

If not, add its folder to the path using the `addpath` function:

```
addpath('D:\user5\matlab\mywork')
```

Now you can run the program just by typing the name of the file at the MATLAB command prompt:

```
drawPlot(xdata, ydata)
```

Getting the Bugs Out

In all but the simplest programs, you are likely to encounter some type of unexpected behavior when you run the program for the first time. Program defects can show up in the form of warning or error messages displayed in the command window, programs that hang (never terminate), inaccurate results, or some number of other symptoms. This is where the second functionality of the MATLAB Editor/Debugger becomes useful.

The MATLAB Debugger enables you to examine the inner workings of your program while you run it. You can stop the execution of your program at any point and then continue from that point, stepping through the code line by line and examining the results of each operation performed. You have the choice of operating the debugger from the Editor window that displays your program, from the MATLAB command line, or both.

The Debugging Process

You can step through the program right from the start if you want. For longer programs, you will probably save time by stopping the program somewhere in the middle and stepping through from there. You can do this by approximating where the program code breaks and setting a stopping point (or *breakpoint*) at that line. Once a breakpoint has been set, start your program from the MATLAB command prompt. MATLAB opens an Editor/Debugger window (if it is not already open) showing a green arrow pointing to the next line to execute.

From this point, you can examine any values passed into the program, or the results of each operation performed. You can step through the program line by line to see which path is taken and why. You can step into any functions that your program calls, or choose to step over them and just see the end results. You can also modify the values assigned to a variable and see how that affects the outcome.

To learn about using the MATLAB Debugger, see "Debugging Process and Features" in the Desktop Tools and Development Environment documentation. Type `help debug` for a listing of all MATLAB debug functions.

For programming tips on how to debug, see "Debugging" on page 10-21 in the Programming Tips documentation.

Cleaning Up the Program

Even after your program is bug-free, there are still some steps you can take to improve its performance and readability. The MATLAB Code Analyzer utility generates a report that can highlight potential problems in your code. For example, you might be using the element-wise AND operator (`&`) where the short-circuit AND (`&&`) is more appropriate. You might be using the `find` function in a context where logical subscripting would be faster.

MATLAB offers the Code Analyzer and several other reporting utilities to help you make the finishing touches to your program code. These tools are described under "Tuning and Managing MATLAB Code Files"in the Desktop Tools and Development Environment documentation.

Improving Performance

The MATLAB Profiler generates a report that shows how your program spends its processing time. For details about using the MATLAB Profiler, see Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation. For tips on other ways to improve the performance of your programs, see Chapter 8, "Performance" in the Programming Fundamentals documentation.

Three types of reports are available:

- "Summary Report" on page 3-5
- "Detail Report" on page 3-5
- "File Listing" on page 3-6

Summary Report

The summary report provides performance information on your main program and on every function it calls. This includes how many times each function is called, the total time spent in that function, along with a bar graph showing the relative time spent by each function.

Detail Report

When you click a function name in the summary report, MATLAB displays a detailed report on that function. This report shows the lines of that function

that take up the most time, the time spent executing that line, the percentage of total time for that function that is spent on that line, and a bar graph showing the relative time spent on the line.

File Listing

The detail report for a function also displays all code for that function. This listing enables you to view the time-consuming code in the context of the entire function body. For every line of code that takes any significant time, additional performance information is provided by the statistics and by the color and degree of highlighting of the program code.

Checking It In

Source control systems offer a way to manage large numbers of files while they are under development. They keep track of the work done on these files as your project progresses, and also ensure that changes are made in a secure and orderly fashion.

If you have a source control system available, you will probably want to check your files into the system once they are complete. If further work is required on one of those files, you just check it back out, make the necessary modifications, and then check it back in again.

MATLAB provides an interface to external source control systems so that you can check files in and out directly from your MATLAB session. See Revision Control in the Desktop Tools and Development Environment documentation for instructions on how to use this interface.

Protecting Your Source Code

Although MATLAB source (`.m`) code is executable by itself, the contents of MATLAB source files are easily accessed, revealing design and implementation details. If you do not want to distribute your proprietary application code in this format, you can use one of these more secure options instead:

- Deploy as P-code — Convert some or all of your source code files to a content-obscured form called a *P-code* file (from its `.p` file extension), and distribute your application code in this format.

- Compile into binary format — Compile your source code files using the MATLAB Compiler to produce a standalone application. Distribute the latter to end users of your application.

In general, if you want to run the code as a standalone application outside of MATLAB, it is best to use the MATLAB® Compiler™ to make your code secure . If you plan to run the code within the MATLAB environment, there is no need to run the Compiler. Instead, convert to P-code those modules of your source code that need to be secure.

Building a Content Obscured Format with P-Code

A P-code file behaves the same as the MATLAB source from which it was produced. The P-code file also runs at the same speed as the source file. Because the contents of P-code files are purposely obscured, they offer a secure means of distribution outside of your organization.

Note Because users of P-code files cannot view the MATLAB code, consider providing diagnostics to enable a user to proceed in the event of an error.

Building the P-Code File. To generate a P-code file, enter the following command in the MATLAB Command Window:

```
pcode file1 file2, ...
```

The command produces the files, file1.p, file2.p, and so on. To convert *all* .m source files residing in your current folder to P-code files, use the command:

```
pcode *.m
```

See the pcode function reference page for a description of all syntaxes for generating P-code files.

Invoking the P-Code File. You invoke the resulting P-code file in the same way you invoke the MATLAB .m source file from which it was derived. For example, to invoke file myfun.p, type

```
[out, out2, ...] = myfun(in1, in2, ...);
```

To invoke script `myscript.p`, type

```
myscript;
```

When you call a P-code file, MATLAB gives it execution precedence over its corresponding `.m` source file. This is true even if you happen to change the source code at some point after generating the P-code file. Remember to remove the `.m` source file before distributing your code.

Running Older P-Code Files on Later Versions of MATLAB. P-Code files are designed to be independent of the release under which they were created and the release in which they are used (backward and forward compatibility). New and deprecated MATLAB features can be a problem, but it is the same problem that would exist if you used the original MATLAB input file. To fix errors of this kind in a P-code file, fix the corresponding MATLAB input file and create a new P-code file.

P-code files built using MATLAB Version 7.4 and earlier have a different format than those built with more recent versions of MATLAB. You still can use these older P-code files when you run MATLAB 7.4 and later, but this capability could be removed in a future release. MathWorks recommends that you rebuild any P-code files that were built with MATLAB 7.4 or earlier using a more recent version of MATLAB, and then redistribute them as necessary.

Building a Standalone Executable

Another way to protect your source code is to build it into a standalone executable and distribute the executable, along with any other necessary files, to external customers. You must have the MATLAB Compiler and a supported C or C++ compiler installed to prepare files for deployment. The end user, however, does not need MATLAB.

To build a standalone application for your MATLAB application, develop and debug your application following the usual procedure for MATLAB program files. Then, generate the executable file or files following the instructions in "Steps by the developer to Deploy to End Users" in the MATLAB Compiler documentation.

Working with Functions in Files

In this section...
"Overview" on page 3-9
"Types of Program Files" on page 3-9
"Basic Parts of a Program File" on page 3-10
"Creating a Program File" on page 3-15
"Providing Help for Your Program" on page 3-18
"Cleaning Up When the Function Completes" on page 3-18

Overview

The MATLAB software provides a full programming language that enables you to write a series of MATLAB statements into a file and then execute them with a single command. You write your program in an ordinary text file, giving the file a name of `filename.m`. The term you use for `filename` becomes the new command that MATLAB associates with the program. The file extension of `.m` makes this a MATLAB program file.

Types of Program Files

Program files can be *scripts* that simply execute a series of MATLAB statements, or they can be *functions* that also accept input arguments and produce output.

MATLAB scripts:

- Are useful for automating a series of steps you need to perform many times.
- Do not accept input arguments or return output arguments.
- Store variables in a workspace that is shared with other scripts and with the MATLAB command line interface.

MATLAB functions:

- Are useful for extending the MATLAB language for your application.

- Can accept input arguments and return output arguments.

- Store variables in a workspace internal to the function.

Basic Parts of a Program File

This simple function shows the basic parts of a program file. Any line that begins with % is not executable:

```
function f = fact(n)                    Function
definition line
% Compute a factorial value.           H1 line
% FACT(N) returns the factorial of N,  Help text
% usually denoted by N!

% Put simply, FACT(N) is PROD(1:N).     Comment
f = prod(1:n);                          Function body
```

This table briefly describes each of these program file parts. Both functions and scripts can have all of these parts, except for the function definition line which applies to functions only. The sections that follow the table describe these parts in greater detail.

File Element	Description
Function definition line (functions only)	Defines the function name, and the number and order of input and output arguments
H1 line	A one line summary description of the program, displayed when you request help on an entire folder, or when you use lookfor
Help text	A more detailed description of the program, displayed together with the H1 line when you request help on a specific function

File Element	Description
Function or script body	Program code that performs the actual computations and assigns values to any output arguments
Comments	Text in the body of the program that explains the internal workings of the program

Function Definition Line

The function definition line informs MATLAB that the file contains a function, and specifies the argument calling sequence of the function. This line contains the function keyword and must always be the first line of the file, except for lines that are nonexecutable comments. The function definition line for the fact function is

All MATLAB functions have a function definition line that follows this pattern.

Function Name. Function names must begin with a letter, can contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed length (returned by the function namelengthmax). Because variables must obey similar rules, you can use the isvarname function to check whether a function name is valid:

```
isvarname myfun
```

Function names also cannot be the same as any MATLAB keyword. Use the `iskeyword` function with no inputs to display a list of all keywords.

Although function names can be of any length, MATLAB uses only the first `N` characters of the name (where `N` is the number returned by the function `namelengthmax`) and ignores the rest. Hence, it is important to make each function name unique in the first `N` characters:

```
N = namelengthmax
N =
    63
```

Note Some operating systems might restrict file names to shorter lengths.

The name of the text file that contains a MATLAB function consists of the function name with the extension `.m` appended. For example,

```
average.m
```

If the file name and the function definition line name are different, MATLAB ignores the internal (function) name. Thus, if `average.m` is the file that defines a function named `computeAverage`, you would invoke the function by typing

```
average
```

Note While the function name specified on the function definition line does not have to be the same as the file name, it is best to use the same name for both to avoid confusion.

Function Arguments. If the function has multiple output values, enclose the output argument list in square brackets. Input arguments, if present, are enclosed in parentheses following the function name. Use commas to separate multiple input or output arguments. Here is the declaration for a function named `sphere` that has three inputs and three outputs:

```
function [x, y, z] = sphere(theta, phi, rho)
```

If there is no output, leave the output blank

```
function printresults(x)
```

or use empty square brackets:

```
function [] = printresults(x)
```

The variables that you pass to the function do not need to have the same name as the variables in the function definition line.

The H1 Line

The H1 line, so named because it is the first help text line, is a comment line immediately following the function definition line. Because it consists of comment text, the H1 line begins with a percent sign, %. For the `average` function, the H1 line is

```
% AVERAGE Mean of vector elements.
```

This is the first line of text that appears when a user types `help` *functionname* at the MATLAB prompt. Further, the `lookfor` function searches on and displays only the H1 line. Because this line provides important summary information about the file, it is important to make it as descriptive as possible.

Help Text

You can create online help for your program files by entering help text on one or more consecutive comment lines at the start of your program. MATLAB considers the first group of consecutive lines immediately following the H1 line that begin with % to be the online help text for the function. The first line without % as the left-most character ends the help.

The help text for the `average` function is

```
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
```

When you type help *functionname* at the command prompt, MATLAB displays the H1 line followed by the online help text for that function. The help system ignores any comment lines that appear after this help block.

Note Help text in a program file can be viewed at the MATLAB command prompt only (using help functionname). You cannot display this text using the MATLAB Help browser. You can, however, use the Help browser to get help on MATLAB functions and also to read the documentation on any MathWorks products.

The Function or Script Body

The function body contains all the MATLAB code that performs computations and assigns values to output arguments. The statements in the function body can consist of function calls, programming constructs like flow control and interactive input/output, calculations, assignments, comments, and blank lines.

For example, the body of the average function contains a number of simple programming statements:

```
[m,n] = size(x);
if (~((m == 1) || (n == 1)) || ...
    (m == 1 && n == 1))  % Flow control

    error('Input must be a vector') % Error message display
end
y = sum(x)/length(x);             % Computation and assignment
```

Comments

As mentioned earlier, comment lines begin with a percent sign (%). Comment lines can appear anywhere in a program file, and you can append comments to the end of a line of code. For example,

```
% Add up all the vector elements.
y = sum(x)           % Use the sum function.
```

In addition to comment lines, you can insert blank lines anywhere in the file. Blank lines are ignored. However, a blank line can indicate the end of the help text entry for a program file.

Block Comments. To write comments that require more than one line, use the block comment operators, `%{` and `%}`:

```
%{
This next block of code checks the number of inputs
passed in, makes sure that each input is a valid data
type, and then branches to start processing the data.
%}
```

Note The `%{` and `%}` operators must appear alone on the lines that immediately precede and follow the block of help text. Do not include any other text on these lines.

Creating a Program File

You create files for your programs using a text editor. MATLAB provides a built-in editor, but you can use any text editor you like. Once you have written and saved the file, you can run the program as you would any other MATLAB function or command.

The process looks like this:

1 Create an M-file using a text editor.

```
function c = myfile(a,b)
c = sqrt((a.^2)+(b.^2))
```

2 Call the M-file from the command line, or from within another M-file.

```
a = 7.5
b = 3.342
c = myfile(a,b)

c =

    8.2109
```

Using Text Editors

Program files are ordinary text files that you create using a text editor. If you use the MATLAB Editor/Debugger, open a new file by selecting **New > File** from the **File** menu at the top of the MATLAB Command Window.

Another way to edit a program file is from the MATLAB command line using the edit function. For example,

```
edit foo
```

opens the editor on the file foo.m. Omitting a file name opens the editor on an untitled file.

You can create the fact function shown in "Basic Parts of a Program File" on page 3-10 by opening your text editor, entering the lines shown, and saving the text in a file called fact.m in your current folder.

Once you have created this file, here are some things you can:

- List the names of the files in your current folder:

```
what
```

- List the contents of file `fact.m`:

```
type fact
```

- Call the `fact` function:

```
fact(5)
ans =
      120
```

A Word of Caution on Saving Program Files

Save any files you create and any MathWorks supplied files that you edit in folders outside of the folder tree in which the MATLAB software is installed. If you keep your files in any of the installed folders, your files can be overwritten when you install a new version of MATLAB.

MATLAB installs its software into folders under *matlabroot*/toolbox. To see what *matlabroot* is on your system, type matlabroot at the MATLAB command prompt.

Also note that locations of files in the *matlabroot*/toolbox folder tree are loaded and cached in memory at the beginning of each MATLAB session to improve performance. If you save files to *matlabroot*/toolbox folders using an external editor, or if you add or remove files from these folders using file system operations, enter the commands clear functionname and rehash toolbox before you use the files in the current session.

For more information, see the rehash function reference page or the section Toolbox Path Caching in the Desktop Tools and Development Environment documentation.

Providing Help for Your Program

You can provide user information for the programs you write by including a help text section at the beginning of your program file. (See "Help Text" on page 3-13).

You can also make help entries for an entire folder by creating a file with the special name `Contents.m` that resides in the folder. This file must contain only comment lines; that is, every line must begin with a percent sign. MATLAB displays the lines in a `Contents.m` file whenever you type

```
help foldername
```

`Contents.m` files are optional. If you have a folder that is on the path that does not contain a `Contents.m` file, MATLAB displays the first comment line (the "H1" line) of each `.m` file in response to typing `help foldername`. If you do not want to display any help summaries at all, create an empty an empty `Contents.m` file in that folder. When an empty `Contents.m` file exists, typing `help foldername` causes MATLAB to respond with `No help found for foldername.`.

To get help in creating and validating your `Contents.m` files, you can use the Contents Report tool in the Current Folder browser. See "Displaying and Updating a Report on the Contents of a Folder" in the MATLAB Desktop Tools and Development Environment documentation for more information.

Cleaning Up When the Function Completes

When you have programmed all that you set out to do in your file, there is one last step to consider before it is complete. Make sure that you leave your program environment in a clean state that does not interfere with any other program code. For example, you might want to

• Close any files that you opened for import or export.

• Restore the MATLAB path.

• Lock or unlock memory to prevent or allow erasing MATLAB function or MEX-files.

• Set your working folder back to its default if you have changed it.

• Make sure global and persistent variables are in the correct state.

MATLAB provides the `onCleanup` function for this purpose. This function, when used within any program, establishes a cleanup routine for that function. When the function terminates, whether normally or in the event of an error or **Ctrl+C**, MATLAB automatically executes the cleanup routine.

The following statement establishes a cleanup routine `cleanupFun` for the currently running program:

```
cleanupObj = onCleanup(@cleanupFun);
```

When your program exits, MATLAB finds any instances of the `onCleanup` class and executes the associated function handles. The process of generating and activating function cleanup involves the following steps:

1 Write one or more cleanup routines for the program under development. Assume for now that it takes only one such routine.

2 Create a function handle for the cleanup routine.

3 At some point, generally early in your program code, insert a call to the `oncleanup` function, passing the function handle.

4 When the program is run, the call to `onCleanup` constructs a cleanup object that contains a handle to the cleanup routine created in step 1.

5 When the program ends, MATLAB implicitly clears all objects that are local variables. This invokes the destructor method for each local object in your program, including the cleanup object constructed in step 4.

6 The destructor method for this object invokes this routine if it exists. This perform the tasks needed to restore your programming environment.

You can declare any number of cleanup routines for a program file. Each call to `onCleanup` establishes a separate cleanup routine for each cleanup object returned.

If, for some reason, the object returned by `onCleanup` persists beyond the life of your program, then the cleanup routine associated with that object is not run when your function terminates. Instead, it will run whenever the object is destroyed (e.g., by clearing the object variable).

Your cleanup routine should never rely on variables that are defined outside of that routine. For example, the nested function shown here on the left executes with no error, whereas the very similar one on the right fails with the error, Undefined function or variable 'k'. This results from the cleanup routine's reliance on variable k which is defined outside of the nested cleanup routine:

```
function testCleanup                function testCleanup
k = 3;                              k = 3;
myFun                               obj = onCleanup(@myFun);
    function myFun                      function myFun
    fprintf('k is %d\n', k)             fprintf('k is %d\n', k)
    end                                 end
end                                 end
```

Examples of Cleaning Up a Program Upon Exit

Example 1 — Close Open Files on Exit. MATLAB closes the file with identifier fid when function openFileSafely terminates:

```
function openFileSafely(fileName)
fid = fopen(fileName, 'r');
c = onCleanup(@()fclose(fid));

s = fread(fid);
    .
    .
    .
end
```

Example 2 — Maintain the Selected Folder. This example preserves the current folder whether functionThatMayError returns an error or not:

```
function changeFolderSafely(fileName)
   currentFolder = pwd;
   c = onCleanup(@()cd(currentFolder));

   functionThatMayError;
   end    % c executes cd(currentFolder) here.
```

Example 3 — Close Figure and Restore MATLAB Path. This example
extends the MATLAB path to include files in the toolbox\images folders, and
then displays a figure from one of these folders. After the figure displays,
the cleanup routine restore_env closes the figure and restores the path to
its original state:

```
function showImageOutsidePath(imageFile)
fig1 = figure;
imgpath = genpath([matlabroot '\toolbox\images']);

% Define the cleanup routine.
cleanupObj = onCleanup(@()restore_env(fig1, imgpath));

% Modify the path to gain access to the image file,
% and display the image.
addpath(imgpath);
rgb = imread(imageFile);
fprintf('\n   Opening the figure %s\n', imageFile);
image(rgb);
pause(2);

    % This is the cleanup routine.
    function restore_env(fighandle, newpath)
    disp '   Closing the figure'
    close(fighandle);
    pause(2)

    disp '   Restoring the path'
    rmpath(newpath);
    end
end
```

Run the function as shown here. You can verify that the path has been
restored by comparing the length of the path before and after running the
function:

```
origLen = length(path);

showImageOutsidePath('greens.jpg')
   Opening the figure greens.jpg
   Closing the figure
```

```
    Restoring the path

currLen = length(path);
currLen == origLen
ans =
     1
```

Retrieving Information About the Cleanup Routine

In Example 3 shown above, the cleanup routine and data needed to call it are contained in a handle to an anonymous function:

```
@()restore_env(fig1, imgpath)
```

The details of that handle are then contained within the object returned by the onCleanup function:

```
cleanupObj = onCleanup(@()restore_env(fig1, imgpath));
```

You can access these details using the task property of the cleanup object as shown here. (Modify the showImageOutsidePath function by adding the following code just before the comment line that says, "% This is the cleanup routine.")

```
disp '   Displaying information from the function handle:'
task = cleanupObj.task;
fun = functions(task)
wsp = fun.workspace{2,1}
fprintf('\n');
pause(2);
```

Run the modified function to see the output of the functions command and the contents of one of the workspace cells:

```
showImageOutsidePath('greens.jpg')

Opening the figure greens.jpg
Displaying information from the function handle:
fun =
    function: '@()restore_env(fig1,imgpath)'
        type: 'anonymous'
```

```
            file: 'c:\work\g6.m'
       workspace: {2x1 cell}
  wsp =
        imageFile: 'greens.jpg'
             fig1: 1
          imgpath: [1x3957 char]
       cleanupObj: [1x1 onCleanup]
              rgb: [300x500x3 uint8]
             task: @()restore_env(fig1,imgpath)

  Closing the figure
  Restoring the path
```

Using onCleanup Versus try-catch

Another way to run a cleanup routine when a function terminates
unexpectedly is to use a try-catch statement. There are limitations to using
this technique however. If the user ends the program by typing **Ctrl+C**,
MATLAB immediately exits the try block, and the cleanup routine never
executes. The cleanup routine also does not run when you exit the function
normally.

The following program cleans up if an error occurs, but not in response to
Ctrl+C:

```
function cleanupByCatch
try
    pause(10);
catch
    disp('   Collecting information about the error')
    disp('   Executing cleanup tasks')
end
```

Unlike the try-catch statement, the onCleanup function responds not only
to a normal exit from your program and any error that might be thrown, but
also to **Ctrl+C**. This next example replaces the try-catch with onCleanup:

```
function cleanupByFunc
obj = onCleanup(@()...
    disp('   Executing cleanup tasks'));
```

```
pause(10);
```

onCleanup in Scripts

onCleanup does not work in scripts as it does in functions. In functions, the cleanup object is stored in the function workspace. When the function exits, this workspace is cleared thus executing the associated cleanup routine. In scripts, the cleanup object is stored in the base workspace (that is, the workspace used in interactive work done at the command prompt). Because exiting a script has no effect on the base workspace, the cleanup object is not cleared and the routine associated with that object does not execute. To use this type of cleanup mechanism in a script, you would have to explicitly clear the object from the command line or another script when the first script terminates.

Scripts and Functions

In this section...

Scripts

Scripts are the simplest kind of program file because they have no input or output arguments. They are useful for automating series of MATLAB commands, such as computations that you have to perform repeatedly from the command line.

The Base Workspace

Scripts share the base workspace with your interactive MATLAB session and with other scripts. They operate on existing data in the workspace, or they can create new data on which to operate. Any variables that scripts create remain in the workspace after the script finishes so you can use them for further computations. Be aware, though, that running a script can unintentionally overwrite data stored in the base workspace by commands entered at the MATLAB command prompt.

Simple Script Example

These statements calculate rho for several trigonometric functions of theta, then create a series of polar plots:

```
% A script to produce        % Comment lines
% "flower petal" plots
theta = -pi:0.01:pi;                % Computations
rho(1,:) = 2 * sin(5 * theta) .^ 2;
rho(2,:) = cos(10 * theta) .^ 3;
rho(3,:) = sin(theta) .^ 2;
rho(4,:) = 5 * cos(3.5 * theta) .^ 3;
```

```
for k = 1:4
    polar(theta, rho(k,:))          % Graphics output
    pause
end
```

Try entering these commands in a file called `petals.m`. This file is now a MATLAB script. Typing `petals` at the MATLAB command line executes the statements in the script.

After the script displays a plot, press **Enter** or **Return** to move to the next plot. There are no input or output arguments; `petals` creates the variables it needs in the MATLAB workspace. When execution completes, the variables (`k`, `theta`, and `rho`) remain in the workspace. To see a listing of them, enter `whos` at the command prompt.

Functions

The main difference between a script and a function is that a function accepts input from and returns output to its caller, whereas scripts do not. You define MATLAB functions in a file that begins with a line containing the `function` key word. You cannot define a function within a script file or at the MATLAB command line.

Functions always begin with a function definition line and end either with the first matching `end` statement, the occurrence of another function definition line, or the end of the file, whichever comes first. Using `end` to mark the end of a function definition is required only when the function being defined contains one or more nested functions.

Functions operate on variables within their own workspace. This workspace is separate from the base workspace; the workspace that you access at the MATLAB command prompt and in scripts.

The Function Workspace

Each function in a file has an area of memory, separate from the MATLAB base workspace, in which it operates. This area, called the function workspace, gives each function its own workspace context.

While using MATLAB, the only variables you can access are those in the calling context, be it the base workspace or that of another function. The variables that you pass to a function must be in the calling context, and the function returns its output arguments to the calling workspace context. You can, however, define variables as global variables explicitly, allowing more than one workspace context to access them. You can also evaluate any MATLAB statement using variables from either the base workspace or the workspace of the calling function using the evalin function. See "Extending Variable Scope" on page 2-17 for more information.

Simple Function Example

The average function is a simple file that calculates the average of the elements in a vector:

```
function y = average(x)
% AVERAGE Mean of vector elements.
% AVERAGE(X), where X is a vector, is the mean of vector
% elements. Nonvector input results in an error.
[m,n] = size(x);
if (~((m == 1) | (n == 1)) | (m == 1 & n == 1))
    error('Input must be a vector')
end
y = sum(x)/length(x);      % Actual computation
```

Try entering these commands in a file called average.m. The average function accepts a single input argument and returns a single output argument. To call the average function, enter

```
z = 1:99;

average(z)
ans =
    50
```

Types of Functions

MATLAB provides the following types of functions. Each function type is described in more detail in a later section of this documentation:

- The primary function is the first function in a program file and typically contains the main program.

- Subfunctions act as subroutines to the main function. You can also use them to define multiple functions within a single file.

- Nested functions are functions defined within another function. They can help to improve the readability of your program and also give you more flexible access to variables in the file.

- Anonymous functions provide a quick way of making a function from any MATLAB expression. You can compose anonymous functions either from within another function or at the MATLAB command prompt.

- Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. They are similar to overloaded functions in any object-oriented language.

- Private functions give you a way to restrict access to a function. You can call them only from a function in the parent folder.

You might also see the term function functions in the documentation. This is not really a separate function type. The term function functions refers to any functions that accept another function as an input argument. You can pass a function to another function using a function handle.

Organizing Your Functions

When writing and saving your functions, you have several options on how to organize the functions within the file, and also where in your folder structure you want to save them. Be sure to place your function files either in the folder in which you plan to run MATLAB, or in some other folder that is on the MATLAB path.

Use this table as a general guide when creating and saving your files:

If your program or routine . . .	then . . .
Requires only one function	Make it a single (primary) function in the file.
Also requires subroutines	Make each subroutine a subfunction within same file as the primary.

If your program or routine . . .	then . . .
Is for use only in the context of a certain function	Nest it within the other function. Nested functions also offer wider access to variables within the function.
Is a constructor or method of a MATLAB class	Put the file in a MATLAB class folder.
Is to have limited access	Put the file in a private subfolder.
Is part of a group of similar functions or classes	Put the file in a package subfolder.

If necessary, you can work around some of the constraints regarding function access by using function handles. You might find this useful when debugging your functions.

Identifying Dependencies

If you need to know what other functions and scripts your program is dependent upon, use one of the techniques described below.

Simple Display of Program File Dependencies

For a simple display of all program files referenced by a particular function, follow these steps:

1 Type `clear functions` to clear all functions from memory (see Note below).

> **Note** `clear functions` does not clear functions locked by `mlock`. If you have locked functions (which you can check using `inmem`) unlock them with `munlock`, and then repeat step 1.

2 Execute the function you want to check. Note that the function arguments you choose to use in this step are important, because you can get different results when calling the same function with different arguments.

3 Type `inmem` to display all program files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output:

```
[mfiles, mexfiles] = inmem
```

Detailed Display of Program File Dependencies

For a much more detailed display of dependent function information, use the `depfun` function. In addition to program files, `depfun` shows which built-ins and classes a particular function depends on:

```
[list, builtins, classes] = depfun('strtok.m');

list
list =
    'D:\matlabR14\toolbox\matlab\strfun\strtok.m'
    'D:\matlabR14\toolbox\distcomp\toChar.m'
    'D:\matlabR14\toolbox\matlab\datafun\prod.m'
    'D:\matlabR14\toolbox\matlab\datatypes\@opaque\char.m'
              .
              .
              .
```

Calling Functions

In this section...

Command vs. Function Syntax

Overview

MATLAB syntax differs for issuing commands and for calling functions. This is sometimes referred to as *command-function duality*. In many cases, you can use either of the two syntaxes in commands and in function calls.

You are issuing a command when you tell MATLAB to do something. You might use a function name in the command and possibly a few switches or modifiers, usually expressed in the form of character strings. But, in most cases, there is no need to pass data such as numeric values or variables in a command. There is also no expectation that the command returns any output value.

A few examples of MATLAB commands are shown here. Note that MATLAB passes only string arguments and does not return any output:

```
clear
format long
dbstop in find_maxval
addpath C:\srcfolder D:\matlab\testfolder2 -end
whos -file savefile.mat -regexp ^p*
```

Unlike commands, most functions act upon data. You pass data into the function when calling it, and often expect it to return data upon completion.

These input and output values can be in the form of any MATLAB class (e.g., arrays of integers, characters, cells, function handles, etc.).

Examples of function calls are shown below. Note that MATLAB requires parentheses around the input argument list, quotation marks around characters and strings, and brackets around multiple output values:

```
y = eye(n)
C = complex(A, B);
m = memmapfile('records.dat', 'Offset', 1024, ...
    'Format', 'uint32');
[x1 x2 x3 x4] = deal(A{:});
```

Regardless of which syntax you use, function names are always sensitive to case. When you call a function, use the correct combination of upper and lowercase letters so that the name is an exact match. Otherwise, you risk calling a different function that does match, but is elsewhere on the path.

MATLAB Command Syntax

A command that makes a function call consists of the function name followed by one or more arguments separated by spaces. In most cases, all input arguments are considered to be strings. Because of this, enclosing string arguments with quotation marks is optional.

The format for calling a function using command syntax is

```
functionname string1 string2 string3
```

You can use command syntax in calling a function when both of the following are true:

- All input arguments must be characters or strings. Variable names, expressions that require evaluation, and non-character classes (e.g., `doubles`, `structures`, `function handles`) are not allowed.

- You do not need to capture output in a variable. Commands that display information on your monitor screen are acceptable.

Note There are a few functions, such as `save` and `load`, that might appear to depart from this first rule. For an explanation of how these functions operate when using this syntax, see "Using save and load with Command Syntax" on page 3-34.

When you use command syntax, MATLAB interprets each input argument as a character string literal. There is no need to enclose these string arguments in quotation marks unless the argument includes one or more space characters. This is true whether the argument is a string of plain text, a file name, or a command switch:

```
strcat one two three four        % Command with 4 arguments.
ans =
   onetwothreefour

strcat 'one' 'two' 'three' 'four'  % Command with 4 arguments.
ans =
   onetwothreefour

strcat 'one two three four'       % Command with 1 argument.
ans =
   one two three four
```

Several examples of command syntax are given below.

Example 1. This command copies file `square.m` to folder `D:\matlab\functions`. All arguments are strings:

```
copyfile square.m D:\matlab\functions
```

Example 2. The example on the left calls `disp` using command syntax. MATLAB interprets A as a string literal and displays the character A.

The example on the right passes the *value* of A to `disp`, which then displays 3.1416:

Command Syntax	Function Syntax
`A = pi;`	`A = pi;`
`disp A` `    A`	`disp(A)` `    3.1416`

Using save and load with Command Syntax. There are a few functions, such as save and load, that do accept variable names as input arguments. Examples follow:

```
save mydata.mat x y z       % x, y, and z are variables
load mydata x z             % x and z are variables
clear N                     % N is a variable
whos A                      % A is a variable
```

MATLAB Function Syntax

Function calls written in function syntax

- Enclose the input argument list in parentheses

- Separate the inputs with commas

- Enclose string arguments with single quotation marks

- Optionally assign any output from the function to one or more output arguments

Unlike command syntax, there are no limitations on when you can use function syntax in a function call.

Function calls written in function syntax enclose the input argument list in parentheses, separate the inputs with commas, enclose string arguments with single quotation marks, and optionally assign any output from the function to one or more output arguments. Unlike command syntax, there are no limitations on when you can use function syntax in a function call.

The format for MATLAB function syntax is

```
out = functionname(variable, 'string', expression, ...);
```

Calls written in function syntax pass the *values* assigned to each variable in the argument list. For example, this expression passes the values assigned to A0, A1, and A2 to the polyeig function:

```
e = polyeig(A0, A1, A2)
```

If a function returns more than one value, separate the output variables with commas or spaces, and enclose them all in square brackets ([ ]):

```
[out1, out2, ..., outN] = functionname(in1, in2, ..., inN);
```

For example,

```
[pathstr, name, ext] = fileparts(filename);
```

Several examples of function syntax appear below. For more examples, see the section on "Common Mistakes In Syntax" on page 3-36

Example 1 — Simple Variable Comparison. Passing two variables representing equal strings to the strcmp function using function and command syntaxes gives different results. The function syntax passes the values of the arguments. strcmp returns a 1, which means they are equal:

```
str1 = 'one';        str2 = 'one';

strcmp(str1, str2)                    % Function syntax
ans =
   1        (equal)
```

The command syntax passes the names of the variables, 'str1' and 'str2', which are unequal:

```
str1 = 'one';        str2 = 'one';

strcmp str1 str2                      % Command syntax
ans =
   0        (unequal)
```

Example 2 — Passing Variable Names. The reshape function takes three input arguments: a variable name and two integers to specify dimensions for the new shape. It also returns the reshaped array. For both of these reasons, you need to use function syntax for this operation:

```
S1 = ...
   'MATLAB: Accelerating the pace of engineering and science.';

S2 = reshape(S1, 19, 3);
S2'
ans =
MATLAB: Acceleratin
g the pace of engin
eering and science.
```

Command syntax interprets all three input arguments as strings and provides no means for capturing the output:

```
reshape S1 19 3;
??? Error using ==> reshape
Size arguments must be integer scalars.
```

Common Mistakes In Syntax
The two MATLAB syntax styles are generally easy to use. You should have no difficulty in using them if you keep in mind the rules stated in the previous sections. Just the same, there are certain potential errors to watch out for.

In all examples in this section, it is the group of statements on the left that are incorrect, and the statements on the right that show the correct usage.

Example 1 — Numeric Values Evaluated As Strings. The statement on the left, below, appears to report that 500 is not numeric. However, because this statement uses command syntax, the input is actually the string '500' and not the number. Use function syntax, as shown on the right, to get the correct answer:

```
isnumeric 500              isnumeric(500)
ans =                      ans =
    0                          1
```

Example 2 — Equal Values that Appear As Unequal. In this example, it might seem that MATLAB is reporting the values of variables A and B as unequal. However, it is not the *values* of A and B that are being compared here; it is the variable names 'A' and 'B':

```
A = 500;   B = 500;              A = 500;   B = 500;
isequal A B                      isequal(A, B)
ans =                            ans =
     0                                1
```

Example 3 — Command Switches Used in Function Syntax. When using a command switch or modifier with function syntax, remember to enclose not only the input arguments in quotation marks, but the command switch, as well. In this example, -file needs to have quotation marks around it:

```
whos(-file, 'savefile.mat')       whos('-file', 'tempfile.mat')
```

A simpler method is to use command syntax for this type of statement:

```
whos -file tempfile.mat
```

Example 4 — Translation of Keywords. In command syntax, MATLAB interprets keywords in the same way it does variable names, as string literals. The statement to the left instructs MATLAB to search for a folder with the literal name 'matlabroot', when what was intended was the folder specified by this keyword. Function syntax uses the value of the keyword instead:

```
cd matlabroot                    cd(matlabroot)
```

Example 5 — Variables That Hold File Names. This example, uses the fopen function to open the file accounts.txt. When this is done using command syntax, MATLAB looks for a file named filename, which does not exist. When working with file names that are stored in variables, it is usually best to use function syntax:

```
filename = 'accounts.txt';       filename = 'accounts.txt';
fopen filename;                   fopen(filename);
```

Example 6 — Invalid String Comparisons. This example attempts to see if the class of vector A is an 8-bit unsigned integer (uint8), but the comparison is really between the strings 'class(A)' and 'int8':

```
A = int8(1:8)                 A = int8(1:8)
strcmp class(A) int8          strcmp(class(A), 'int8')
ans =                         ans =
    0                               1
```

Example 7 — Numeric Arguments. This example shows that command syntax does not accept numeric arguments. Because command syntax assumes that each input argument is a character string, the numeric input 3.499 is interpreted by MATLAB as a five-element character array '3.499', numerically equivalent to the vector [51 46 52 57 57].

```
round 3.499                   round(3.499)
ans =                         ans =
   51    46    52    57    57      3
```

Example 8 — Save and Load. The save and load functions are often easier to use with command syntax. The statement save M saves variable M, not the character M, to the workspace:

```
M = magic(20);                M = magic(20);
save(M)                       save M           % or save('M')
clear M                       clear M
load(M)                       load M           % or load('M')
```

Example 9 — Class as a Command. When using the class function to display or return the class of a variable or value, always use the function syntax:

```
class pi                      class(pi)
ans =                         ans =
   char                          double
```

Recognizing Function Calls That Use Command Syntax

It can be difficult to tell whether a MATLAB expression is a function call using command syntax or another kind of expression, such as an operation on one or more variables. Consider the following example:

```
ls ./d
```

Is this a call to the `ls` function with the folder `./d` as its argument? Or is it a request to perform elementwise division on the array that is the value of the `ls` variable, using the value of the `d` variable as the divisor?

This example might appear unambiguous because MATLAB can determine whether `ls` and `d` are functions or variables, but that is not always true. Some MATLAB components, such as the Code Analyzer and the Editor/Debugger, must operate without reference to the MATLAB path or workspace. MATLAB therefore uses syntactic rules to determine when an expression is a function call using command syntax.

The rules are complicated and have exceptions. In general, when MATLAB recognizes an identifier (which might name a function or a variable), it analyzes the characters that follow the identifier to determine what kind of expression exists. The expression is usually a function call using command syntax when all of the following are true:

1 The identifier is followed immediately by white space.

2 The characters following the white space are not parentheses or an assignment operator.

3 The characters following the white space are not an operator that is itself followed by additional white space and then by characters that can legitimately follow an operator.

The example above meets all three criteria and is therefore a function call using command syntax:

```
ls ./d
```

The following examples are not function calls using command syntax:

```
% No white space following the ls identifier
% Interpretation: elementwise division
ls./d

% Parenthesis following white space
% Interpretation: function call using function syntax
```

```
ls ('./d')

% Assignment operator following white space
% Interpretation: assignment to a variable
ls =d

% Operator following white space, followed in turn by
% more white space and a variable
% Interpretation: elementwise division
ls ./ d
```

What Happens When You Call a Function

When you call a function from either the command line or from within another program file, MATLAB parses the function into pseudocode and stores it in memory. This prevents MATLAB from having to reparse a function each time you call it during a session. The pseudocode remains in memory until you clear it using the clear function, or until you quit MATLAB.

Clearing Functions from Memory

You can use clear in any of the following ways to remove functions from the MATLAB workspace.

Syntax	Description
clear functionname	Remove specified function from workspace.
clear functions	Remove all compiled functions.
clear all	Remove all variables and functions.

Determining Which Function Gets Called

When more than one function has the same name, which one does MATLAB call? This section explains the process that MATLAB uses to make this decision. It covers the following topics:

- "Function Scope" on page 3-41

- "Function Precedence Order" on page 3-41

- "Multiple Implementation Types" on page 3-43

• "Querying Which Function Gets Called" on page 3-43

Keep in mind that there are certain situations in which function names can conflict with *variables* of the same name. See "Potential Conflict with Function Names" on page 2-14 for more information.

Function Scope

Any functions you call must first be within the scope of (i.e., visible to) the calling function or your MATLAB session. MATLAB determines if a function is in scope by searching for the function's executable file according to a certain order (see Function Precedence Order, below).

One key part of this search order is the MATLAB path. The path is an ordered list of folders that MATLAB defines on startup. You can add or remove any folders you want from the path. MATLAB searches the path for the given function name, starting at the first folder in the path string and continuing until either the function file is found or the list of folders is exhausted. If no function of that name is found, then the function is considered to be out of scope and MATLAB issues an error.

See "Calling Nested Functions" on page 4-18 in the MATLAB Programming Fundamentals documentation for more information on the scope of nested functions within a MATLAB program file.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. MATLAB selects the correct function for a given context by applying the following function precedence rules in the order given here.

For items 4 through 8 in this list, the file MATLAB searches for can be any of four types: a built-in file or program file with a `.m` file extension, preparsed program file (P-Code), compiled C or Fortran file (MEX-file), or Simulink® model (MDL-file). See "Multiple Implementation Types" on page 3-43 for more on this.

1 "Variables" on page 2-8

Before assuming that a name should match a function, MATLAB checks the current workspace to see if it matches a variable name. If MATLAB finds a match, it stops the search.

2 "Nested Functions" on page 4-16

Nested Functions take precedence over all other functions that are on the path and have the same name.

3 "Subfunctions" on page 4-33

Subfunctions take precedence over all other functions except for nested functions that are on the path and have the same name.

4 "Private Functions" on page 4-35

Private functions are called if there is no subfunction of the same name within the current scope.

5 Class Constructors

Constructor functions (functions having names that are the same as the @ folder, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create a file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

6 Overloaded Methods

Overloaded methods have lower precedence than nested functions, subfunctions, and private functions of the same name that are on the path. This is true even if you call the function with an argument of type matching that of the overloaded method. Which overloaded method gets called depends on the classes of the objects passed in the argument list.

7 Functions in the current folder

A function in the current working folder is selected before one elsewhere on the path.

8 Functions elsewhere on the path

Finally, a function elsewhere on the path is selected. A function in a folder that is toward the beginning of the path string is given higher precedence.

Note Because variables have the highest precedence, if you have created a variable of the same name as a function, MATLAB will not be able to run that function until you clear the variable from memory.

Multiple Implementation Types

There are five file precedence types. MATLAB uses file precedence to select between identically named functions in the same folder. The order of precedence for file types is

1 Built-in file

2 MEX-files

3 MDL (Simulink® model) file

4 P-code file

5 Program file with a `.m` file extension

For example, if MATLAB finds a P-code and a MATLAB program file version of a method in a class folder, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the program file.

Querying Which Function Gets Called

You can determine which function MATLAB will call using the `which` command. For example,

```
which pie3
matlabroot/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
folder_on_your_path/@portfolio/pie3.m       % portfolio method
```

The `which` command determines which version of `pie3` MATLAB calls if you pass a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

Resolving Difficulties In Calling Functions

The two most common problems related to invoking functions in MATLAB are:

- "Conflicting Function and Variable Names" on page 3-44
- "Undefined Functions or Variables" on page 3-44

Conflicting Function and Variable Names

MATLAB throws an error if a variable and function have been given the same name and there is insufficient information available for MATLAB to resolve the conflict. You may see an error message something like the following:

```
Error: <functionName> was previously used as a variable,
  conflicting with its use here as the name of a function
  or command.
```

where `<functionName>` is the name of the function.

Certain uses of the `eval` and `load` functions can also result in a similar conflict between variable and function names. For help in diagnosing the cause of such conflicts, see the sections Avoid Using Function Names for Variables and Potential Conflict with Function Names in the Programming Fundamentals documentation.

Undefined Functions or Variables

You may encounter the following error message, or something similar, while working with functions or variables in MATLAB:

```
??? Undefined function or variable 'x'.
```

These errors usually indicate that MATLAB cannot find a particular variable or MATLAB program file in the current directory or on the search path. The root cause is likely to be one of the following:

- The name of the function has been misspelled.
- The function name and name of the file containing the function are not the same.
- The toolbox to which the function belongs is not installed.
- The search path to the function has been changed.
- The function is part of a toolbox that you do not have a license for.

Follow the steps described in this section to resolve this situation.

Verify that You Have the Correct Spelling of the Function Name. One of the first things to check when you are unable to invoke a function is the spelling of the function name. Especially with longer function names or names containing similar characters (e.g., letter l and numeral one), it is easy to make an error that is not easily detected.

An unrecognized function name may also be the result of case mismatch. For example, the name of the MATLAB function accumarray contains lowercase letters only. The following command fails because it includes an uppercase letter in the function name:

```
accumArray
??? Undefined function or variable 'accumArray'.
```

Using the alphabetical or categorized function lists in the MATLAB Help Browser can help you find the correct spelling.

Make Sure the Function Name Matches the File Name. You establish the name for a function when you write its function definition line. This name should always match the name of the file you save it to. For example. if you create a function named curveplot,

```
function curveplot(xVal, yVal)
     - program code -
```

then you should name the file containing that function `curveplot.m`. If you create a `pcode` file for the function, then name that file `curveplot.p`. In the case of conflicting function and file names, the file name overrides the name given to the function. In this example, if you save the `curveplot` function to a file named `curveplotfunction.m`, then attempts to invoke the function using the function name will fail:

```
curveplot
??? Undefined function or variable 'curveplot'.
```

If you encounter this problem, change either the function name or file name so that they are the same. If you have difficulty locating the file that uses this function, use the MATLAB **Find Files** utility as follows:

1 Open the **Find Files** dialog box by clicking **Edit > Find Files** in MATLAB.

2 Under **Find files named:** enter `*.m`

3 **Under Find files containing text:** enter the function name.

4 Click the **Find** button
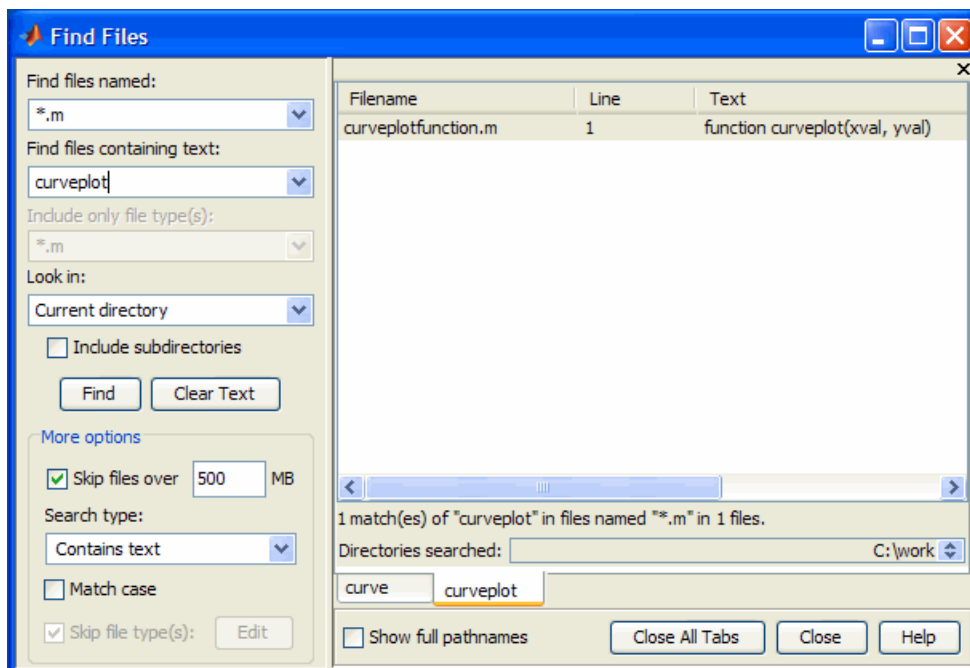
Make Sure the Toolbox Is Installed. If you are unable to use a built-in function from MATLAB or its toolboxes, make sure that the function is installed. If you do not know which toolbox supports the function you need, reference the following list:

   http://www.mathworks.com/support/functions/alpha_list.html

Once you know which toolbox the function belongs to, use the `ver` function to see which toolboxes are installed on the system from which you run MATLAB. The `ver` function displays a list of all currently installed MathWorks products. If you can locate the toolbox you need in the output displayed by `ver`, then the toolbox is installed. For help with installing MathWorks products, see the Installation Guide documentation.

If you do not see the toolbox and you believe that it is installed, then perhaps the MATLAB path has been set incorrectly. Go on to the next section.

Verify the Path Used to Access the Function. This step resets the path to the default. Because MATLAB stores the toolbox information in a cache file, you will need to first update this cache and then reset the path. To do this,

1 Go to the File menu and select **Preferences...**

2 Go to the **General** heading. Click the button **Update Toolbox Path Cache** and press **OK**.

3 Go to the File menu and select **Set Path...**

4 Click **Default**, and a small dialog box opens warning that you will lose your current path settings if you proceed. Click **Yes** if you decide to proceed, and then click **OK** and then **Save** to finish.

(If you have added any custom paths to MATLAB, you will need to restore those later)

Run ver again to see if the toolbox is installed. If not, you may need to reinstall this toolbox to use this function. See the Related Solution 1-1CBD3, "How do I install additional toolboxes into my existing MATLAB" for more information about installing a toolbox.

Once ver shows your toolbox, run the following command to see if you can find the function:

```
which -all <functionname>
```

replacing <functionname> with the name of the function. You should be presented with the path(s) of the function file. If you get a message indicating that the function name was not found, you may need to reinstall that toolbox to make the function active.

Verify that Your License Covers The Toolbox. If you receive the error message "Has no license available", there is a licensing related issue preventing you from using the function. To find the error that is occurring, you can use the following command:

```
license checkout <toolbox_license_key_name>
```

replacing `<toolbox_license_key_name>` with the proper key name for the toolbox that contains your function. To find the license key name, look at the `INCREMENT` lines in your license file. For information on how to find your license file see the related solution: 1-63ZIR6,"Where are the license files for MATLAB located?"

The license key names of all the toolboxes are located after each INCREMENT tag in the `license.dat` file. For example:

```
INCREMENT MATLAB MLM 17 00-jan-0000 0 k
B454554BADECED4258 \HOSTID=123456 SN=123456
```

If your license.dat file has no INCREMENT lines, refer to your license administrator for them. For example, to test the licensing for Symbolic Math Toolbox, you would run the following command:

```
license checkout Symbolic_Toolbox
```

A correct testing gives the result "ANS=1". An incorrect testing results in an error from the license manager. You can either troubleshoot the error by looking up the license manager error here:

```
http://www.mathworks.com/support/install.html
```

or you can contact the Installation Support Team with the error here:

```
http://www.mathworks.com/support/contact_us/index.html
```

When contacting support, provide your license number, your MATLAB version, the function you are using, and the license manager error (if applicable).

Calling External Functions

The MATLAB external interface offers a number of ways to run external functions from MATLAB. This includes programs written in C or Fortran, methods invoked on Sun Java or COM (Component Object Model) objects, functions that interface with serial port hardware, and functions stored in shared libraries. The *MATLAB External Interfaces* documentation describes these various interfaces and how to call these external functions.

Running External Programs

For information on how to invoke operating systems commands or execute programs that are external to MATLAB, see Running External Programs in the MATLAB Desktop Tools and Development documentation.

Function Arguments

In this section...

Overview

When calling a function, the caller provides the function with any data it needs by passing the data in an argument list. Data that needs to be returned to the caller is passed back in a list of return values.

Semantically speaking, the MATLAB software always passes argument data by value. (Internally, MATLAB optimizes away any unnecessary copy operations.) If you pass data to a function that then modifies this data, you will need to update your own copy of the data. You can do this by having the function return the updated value as an output argument.

Input Arguments

The MATLAB software accepts input arguments in either of two different formats. See "Command vs. Function Syntax" on page 3-31 for information on how to use these formats.

Passing String Arguments

When using the function syntax to pass a string literal to a function, you must enclose the string in single quotation marks, ('string'). For example, to create a new folder called myAppTests, use

```
mkdir('myAppTests')
```

However, if you are passing a *variable* to which a string has been assigned, use the variable name alone, without quotation marks. This example passes the variable folderName:

```
folderName = 'myAppTests';
mkdir(folderName)
```

Passing File Name Arguments

You can specify a file name argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The .mat file extension is optional for save and load):

```
load mydata.mat           % Command syntax
load('mydata.mat')        % Function syntax
```

If you assign the output to a variable, you must use the function syntax:

```
savedData = load('mydata.mat')
```

Specify ASCII files as shown here. In this case, the file extension is required:

```
load mydata.dat -ascii           % Command syntax
load('mydata.dat','-ascii')      % Function syntax
```

Determining File Names at Run-Time. There are several ways that your function code can work on specific files without your having to hardcode their file names into the program. You can

- Pass the file name as an argument:

  ```
  function myfun(datafile)
  ```

- Prompt for the file name using the input function:

  ```
  filename = input('Enter name of file:  ', 's');
  ```

- Browse for the file using the uigetfile function:

  ```
  [filename, pathname] = uigetfile('*.mat', 'Select MAT-file');
  ```

Passing Function Handle Arguments

The MATLAB function handle has several uses, the most common being a means of immediate access to the function it represents. You can pass function handles in argument lists to other functions, enabling the receiving function to make calls by means of the handle.

To pass a function handle, include its variable name in the argument list of the call:

```
fhandle = @humps;
x = fminbnd(fhandle, 0.3, 1);
```

The receiving function invokes the function being passed using the usual MATLAB calling syntax:

```
function [xf, fval, exitflag, output] = ...
        fminbnd(fhandle, ax, bx, options, varargin)
            .
            .
            .
113  fx = fhandle(x, varargin{:});
```

Output Arguments

To receive data output from a function, you must use the function calling syntax. This is not supported when you use command calling syntax

Assigning Output Arguments

Use the syntax shown here to store any values that are returned by the function you are calling. To store one output, put the variable that is to hold that output to the left of the equal sign:

```
vout = myfun(vin1, vin2, ...);
```

To store more than one output, list the output variables inside square brackets and separate them with commas or spaces:

```
[vout1 vout2 ...] = myfun(vin1, vin2, ...);
```

The number of output variables in your function call statement does not have to match the number of return values declared in the function being called.

For a function that declares N return values, you can specify anywhere from zero to N output variables in the call statement. Any return values that you do not have an output variable for are discarded.

Functions return output values in the order in which the corresponding output variables appear in the function definition line within the file. This function returns 100 first, then x * y, and lastly x.^2:

```
function [a b c] = myfun(x, y)
b = x * y;    a = 100;    c = x.^2;
```

If called with only one output variable in the call statement, the function returns only 100 and discards the values of b and c. If called with no outputs, the functions returns 100 in the MATLAB default variable ans.

Assigning Optional Return Values

The section "Passing Variable Numbers of Arguments" on page 3-59 describes the method of returning optional outputs in a cell array called varargout. A function that uses varargout to return optional values has a function definition line that looks like one of the following:

```
function varargout = myfun(vin1, vin2, ...)
function [vout1 vout2 ... varargout] = myfun(vin1, vin2, ...)
```

The code within the function builds the varargout cell array. The content and order of elements in the cell array determines how MATLAB assigns optional return values to output variables in the function call.

In the case where varargout is the only variable shown to the left of the equal sign in the function definition line, MATLAB assigns varargout{1} to the first output variable, varargout{2} to the second, and so on. If there are other outputs declared in the function definition line, then MATLAB assigns those outputs to the leftmost output variables in the call statement, and then assigns outputs taken from the varargout array to the remaining output variables in the order just described.

This function builds the varargout array using descending rows of a 5-by-5 matrix. The function is capable of returning up to six outputs:

```
function varargout = byRow(a)
```

```
varargout{1} = '    With VARARGOUT constructed by row ...';
for k = 1:5
    row = 5 - (k-1);                % Reverse row order
    varargout{k+1} = a(row,:);
end
```

Call the function, assigning outputs to four variables. MATLAB returns
varargout{1:4}, with rows of the matrix in varargout{2:4} and in the order
in which they were stored by the function:

```
[text r1 r2 r3] = byRow(magic(5))
text =
    With VARARGOUT constructed by row ...
r1 =
    11    18    25     2     9
r2 =
    10    12    19    21     3
r3 =
     4     6    13    20    22
```

A similar function builds the varargout array using diagonals of a 5-by-5
matrix:

```
function varargout = byDiag(a)
varargout{1} = '    With VARARGOUT constructed by diagonal ...';
for k = -4:4
    varargout{k + 6} = diag(a, k);
end
```

Call the function with five output variables. Again, MATLAB assigns
elements of varargout according to the manner in which it was constructed
within the function:

```
[text d1 d2 d3 d4] = byDiag(magic(5))
text =
    With VARARGOUT constructed by diagonal ...
d1 =
    11
d2 =
    10
    18
```

```
d3 =
       4
      12
      25
d4 =
      23
       6
      19
       2
```

Returning Modified Input Arguments

If you pass any input variables that the function can modify, you will need to include the same variables as output arguments so that the caller receives the updated value.

For example, if the function `readText`, shown below, reads one line of a file each time is it called, then it must keep track of the offset into the file. But when `readText` terminates, its copy of the `offset` variable is cleared from memory. To keep the offset value from being lost, `readText` must return this value to the caller:

```
function [text, offset] = readText(filestart, offset)
```

Passing Arguments in Structures or Cell Arrays

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure or cell array.

Passing Arguments in a Structure

Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields. Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

This example updates weather statistics from information in the following chart.

City	Temp.	Heat Index	Wind Speed	Wind Chill
Boston	43	32	8	37
Chicago	34	27	3	30
Lincoln	25	17	11	16
Denver	15	-5	9	0
Las Vegas	31	22	4	35
San Francisco	52	47	18	42

The information is stored in structure W. The structure has one field for each column of data:

```
W = struct('city', {'Bos','Chi','Lin','Dnv','Vgs','SFr'}, ...
           'temp', {43, 34, 25, 15, 31, 52}, ...
           'heatix', {32, 27, 17, -5, 22, 47}, ...
           'wspeed', {8, 3, 11, 9, 4, 18}, ...
           'wchill', {37, 30, 16, 0, 35, 42});
```

To update the data base, you can pass the entire structure, or just one field with its associated data. In the call shown here, W.wchill is a comma-separated list:

```
updateStats(W.wchill);
```

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The advantage over structures is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function. The disadvantage is that you do not have field names to describe each variable.

This example passes several attribute-value arguments to the plot function:

```
X = -pi:pi/10:pi;
Y = tan(sin(X)) - sin(tan(X));

C{1,1} = 'LineWidth';        C{2,1} = 2;
C{1,2} = 'MarkerEdgeColor';  C{2,2} = 'k';
```

```
C{1,3} = 'MarkerFaceColor';   C{2,3} = 'g';

plot(X, Y, '--rs', C{:})
```

Passing Optional Arguments

When calling a function, there are often some arguments that you must always specify, and others that are optional. For example, the sort function accepts one to three inputs and returns zero to two outputs:

Create a 7-by-7 numeric matrix, A, and sort it along the first dimension in ascending order:

```
A = magic(7);
sort(A, 1, 'ascend');
```

The first input in this call is the matrix to be sorted and is always required. The second and third inputs (dimension and mode, respectively) are optional. If you do not specify a value for either of the optional inputs, MATLAB uses its default value instead. For sort, the default second and third inputs select an ascending sort along the first dimension. If that is the type of sort you intend to do, then you can replace the second command above with the following:

```
sort(A);
```

In the same manner, some output arguments can be optional, as well. In this case, the values for any outputs not specified in the call are simply not returned. The first command shown below returns the sorted matrix in B and the indices used to sort the matrix in ind. The second command returns only the sorted matrix. And the third command returns no values, displaying output to the terminal screen instead:

```
[B, ind] = sort(A);      % Return sorted matrix and indices.
B = sort(A);             % Return sorted matrix.
sort(A);                 % Display output.
```

The optional outputs shown in the last example

You can also ignore outputs when calling a function or ignore certain inputs when writing a function. The command below requests only the matrix of indices (variable ind) from the sort function. The tilde (~) operator tells

MATLAB that the output that holds this position in the argument list is not needed by the caller:

```
[~, ind] = sort(A)
```

Passing Variable Numbers of Arguments

The varargin and varargout functions let you pass any number of inputs or return any number of outputs to a function. This section describes how to use these functions and also covers

- "Unpacking varargin Contents" on page 3-60
- "Packing varargout Contents" on page 3-60
- "varargin and varargout in Argument Lists" on page 3-61

MATLAB packs all specified input arguments into a *cell array*, a special kind of MATLAB array in which the array elements are cells. Each cell can hold any size or kind of data — one might hold a vector of numeric data, another in the same array might hold an array of string data, and so on. For output arguments, your function code must pack them into a cell array so that MATLAB can return the arguments to the caller.

Here is an example function that accepts any number of two-element vectors and draws a line to connect them:

```
function testvar(varargin)
for k = 1:length(varargin)
   x(k) = varargin{k}(1);            % Cell array indexing
   y(k) = varargin{k}(2);
end
xmin = min(0,min(x));
ymin = min(0,min(y));
axis([xmin fix(max(x))+3 ymin fix(max(y))+3])
plot(x,y)
```

Coded this way, the testvar function works with various input lists; for example,

```
testvar([2 3],[1 5],[4 8],[6 5],[4 2],[2 3])
```

```
testvar([-1 0],[3 -5],[4 2],[1 1])
```

Unpacking varargin Contents. Because varargin contains all the input arguments in a cell array, it is necessary to use cell array indexing to extract the data. For example,

```
y(k) = varargin{k}(2);
```

Cell array indexing has two subscript components:

- The indices within curly braces {} specify which cell to get the contents of.

- The indices within parentheses () specify a particular element of that cell.

In the preceding code, the indexing expression {k} accesses the kth cell of varargin. The expression (2) represents the second element of the cell contents.

Packing varargout Contents. When allowing a variable number of output arguments, you must pack all of the output into the varargout cell array. Use nargout to determine how many output arguments the function is called with. For example, this code accepts a two-column input array, where the first column represents a set of x coordinates and the second represents y coordinates. It breaks the array into separate [xi yi] vectors that you can pass into the testvar function shown at the beginning of the section on "Passing Variable Numbers of Arguments" on page 3-59:

```
function [varargout] = testvar2(arrayin)
for k = 1:nargout
   varargout{k} = arrayin(k,:);   % Cell array assignment
end
```

The assignment statement inside the for loop uses cell array assignment syntax. The left side of the statement, the cell array, is indexed using curly braces to indicate that the data goes inside a cell. For complete information on cell array assignment, see the documentation on "Cell Arrays" on page 1-101.

To call testvar2, type

```
a = [1 2; 3 4; 5 6; 7 8; 9 0];
```

```
[p1, p2, p3, p4, p5] = testvar2(a)
p1 =
    1     2
p2 =
    3     4
p3 =
    5     6
p4 =
    7     8
p5 =
    9     0
```

varargin and varargout in Argument Lists. varargin or varargout must appear last in the argument list, following any required input or output variables. That is, the function call must specify the required arguments first. For example, these function declaration lines show the correct placement of varargin and varargout:

```
function [out1,out2] = example1(a,b,varargin)
function [i,j,varargout] = example2(x1,y1,x2,y2,flag)
```

Checking the Number of Input Arguments

The nargin and nargout functions enable you to determine how many input and output arguments a function is called with. You can then use conditional statements to perform different tasks depending on the number of arguments. For example,

```
function c = testarg1(a, b)
if (nargin == 1)
    c = a .^ 2;
elseif (nargin == 2)
    c = a + b;
end
```

Given a single input argument, this function squares the input value. Given two inputs, it adds them together.

Here is a more advanced example that finds the first token in a character string. A *token* is a set of characters delimited by white space or some other

character. Given one input, the function assumes a default delimiter of white space; given two, it lets you specify another delimiter if desired. It also allows for two possible output argument lists:

```
function [token, remainder] = strtok(string, delimiters)
% Function requires at least one input argument
if nargin < 1
   error('Not enough input arguments.');
end
token = []; remainder = [];
len = length(string);
if len == 0
   return
end

% If one input, use white space delimiter
if (nargin == 1)
   delimiters = [9:13 32]; % White space characters
end
i = 1;

% Determine where nondelimiter characters begin
while (any(string(i) == delimiters))
   i = i + 1;
   if (i > len), return, end
end

% Find where token ends
start = i;
while (~any(string(i) == delimiters))
   i = i + 1;
   if (i > len), break, end
end
finish = i - 1;
token = string(start:finish);

% For two output arguments, count characters after
% first delimiter (remainder)
if (nargout == 2)
   remainder = string(finish+1:end);
```

```
    end
```

The `strtok` function is a MATLAB file in the `strfun` folder.

Note The order in which output arguments appear in the function declaration line is important. The argument that the function returns in most cases appears first in the list. Additional, optional arguments are appended to the list.

Passing Optional Arguments to Nested Functions

You can use optional input and output arguments with nested functions, but you should be aware of how MATLAB interprets `varargin`, `varargout`, `nargin`, and `nargout` under those circumstances.

`varargin` and `varargout` are variables and, as such, they follow exactly the same scoping rules as any other MATLAB variable. Because nested functions share the workspaces of all outer functions, `varargin` and `varargout` used in a nested function can refer to optional arguments passed to or from the nested function, or passed to or from one of its outer functions.

`nargin` and `nargout`, on the other hand, are functions and when called within a nested function, always return the number of arguments passed to or from the nested function itself.

Using varargin and varargout. `varargin` or `varargout` used in a nested function can refer to optional arguments passed to or from that function, or to optional arguments passed to or from an outer function.

- If a nested function includes `varargin` or `varargout` in its function declaration line, then the use of `varargin` or `varargout` within that function returns optional arguments passed to or from that function.

- If `varargin` or `varargout` are not in the nested function declaration but are in the declaration of an outer function, then the use of `varargin` or `varargout` within the nested function returns optional arguments passed to the outer function.

In the example below, function C is nested within function B, and function B is nested within function A. The term varargin{1} in function B refers to the second input passed to the primary function A, while varargin{1} in function C refers to the first argument, z, passed from function B:

```
function x = A(y, varargin)    % Primary function A
B(nargin, y * rand(4))

   function B(argsIn, z)       % Nested function B
   if argsIn >= 2
      C(z, varargin{1}, 4.512, 1.729)
   end

      function C(varargin)     % Nested function C
      if nargin >= 2
         x = varargin{1}
      end
      end   % End nested function C
   end   % End nested function B
end   % End primary function A
```

Using nargin and nargout. When nargin or nargout appears in a nested function, it refers to the number of inputs or outputs passed to that particular function, regardless of whether or not it is nested.

In the example shown above, nargin in function A is the number of inputs passed to A, and nargin in function C is the number of inputs passed to C. If a nested function needs the value of nargin or nargout from an outer function, you can pass this value in as a separate argument, as done in function B.

Example of Passing Optional Arguments to Nested Functions. This example references the primary function's varargin cell array from each of two nested functions. (Because the workspace of an outer function is shared with all functions nested within it, there is no need to pass varargin to the nested functions.)

Both nested functions make use of the nargin value that applies to the primary function. Calling nargin from the nested function would return the number of inputs passed to that nested function, and not those that had been

passed to the primary. For this reason, the primary function must pass its
nargin value to the nested functions.

```
function meters = convert2meters(miles, varargin)
% Converts MILES (plus optional FEET and INCHES input)
% values to METERS.

if nargin < 1 || nargin > 3
   error('1 to 3 input arguments are required');
end

   function feet = convert2Feet(argsIn)
   % Nested function that converts miles to feet and adds in
   % optional FEET argument.

   feet = miles .* 5280;

   if argsIn >= 2
      feet = feet + varargin{1};
   end
   end  % End nested function convert2Feet

   function inches = convert2Inches(argsIn)
   % Nested function that converts feet to inches and adds in
   % optional INCHES argument.

   inches = feet .* 12;

   if argsIn == 3
      inches = inches + varargin{2};
   end
   end  % End nested function convert2Inches

feet = convert2Feet(nargin);
inches = convert2Inches(nargin);

meters = inches .* 2.54 ./ 100;
end  % End primary function convert2meters

convert2meters(5)
```

```
ans =
  8.0467e+003

convert2meters(5, 2000, 4.7)
ans =
  8.6564e+003
```

Ignoring Selected Outputs or Input Arguments

Using the tilde (~) operator, you can change the declaration of a function so that it ignores any one or more entries in the input argument list, or you can change the function calling code so that one or more selected values returned by the function are ignored. This can help you to keep your workspace clear of variables you have no use for, and also help you conserve memory.

You can ignore unneeded outputs when calling a function:

```
[vOut1, ~, ~, vOut4] = myTestFun;
```

You can ignore arguments in an input argument list when defining certain functions:

```
function myTestFun(argIn1, ~, argIn3);
```

Note Each tilde operator in an input or output argument list must be separated from other arguments by a comma.

Ignoring Selected Outputs on a Function Call. When you replace an output argument with the tilde operator, MATLAB does not return the value that corresponds to that place in the argument list.

The task performed by the following example is to see if the current working folder contains any files with a .m file extension. The dos function returns up to two outputs: the completion status (where 0 equals success), and the results of the operation. In this example, you only need the status output. Returning the second output adds an unused variable m_files to the workspace, and wastes nearly 50KB of memory:

```
clear
[status, m_files] = dos('dir *.m');
whos
  Name          Size              Bytes  Class      Attributes

  m_files       1x24646           49292  char
  status        1x1                   8  double
```

Replacing the unused variable with the tilde operator resolves these problems:

```
clear
[status, ~] = dos('dir *.m');

status
status =
     0          % Zero status means success: files were found

whos
  Name          Size              Bytes  Class      Attributes

  status        1x1                   8  double
```

The next example displays the names of all files in the current folder that have an xls extension. In the first call to fileparts, only the file extension is needed. In the second call, only the file name is needed. All other outputs are ~, and are thus ignored:

```
s = dir;    len = length(s);
for k=1:len
    [~, ~, fileExt] = fileparts(s(k).name);
    if strcmp(fileExt,'.xls')
        [~, xlsFile] = fileparts(s(k).name)
    end
end

xlsFile =
    my_accounts
xlsFile =
    team_schedule
xlsFile =
    project_goals
```

Ignoring Inputs in a Function Definition. Callback functions and methods that implement a subclass are two examples of how you might find the tilde operator useful when fewer than the full number of *inputs* are required. When writing a callback function, you might have arguments required by the callback template that your own callback code has no use for. Just the same, you need to include these inputs in the argument list for your callback definition. Otherwise, MATLAB might not be able to execute the callback.

The following example callback uses only the first of the three inputs defined in the template. Invoking the callback as it is shown here puts two unused inputs into the function workspace:

```
function plot_popup_CreateFcn(hObject, eventdata, handles)
if ispc && isequal( ...
        get(hObject, 'BackgroundColor'), ...
        get(O,'defaultUicontrolBackground'))
set(hObject,'BackgroundColor','white');
end
```

Rewrite the top line of the callback function so that it replaces the two unused inputs with tilde. MATLAB now ignores any inputs passed in the call. This results in less clutter in the function workspace:

```
function plot_popup_CreateFcn(hObject, ~, ~)
if ispc && isequal( ...
        get(hObject, 'BackgroundColor'), ...
        get(O,'defaultUicontrolBackground'))
set(hObject,'BackgroundColor','white');
end
```

When writing the original callback in the MATLAB Editor, the Code Analyzer utility highlights the second and third input arguments. If you hover your mouse pointer over one of these arguments, the Code Analyzer displays the following warning:

Click the **Fix** button for each of the highlighted arguments, and MATLAB replaces each with the tilde operator:

Another use for tilde as an input argument is in the use of classes that are derived from a superclass. In many cases, inputs required by the superclass might not be needed when invoked on a method of a particular subclass. When writing the subclass method, replace inputs that are unused with tilde. By doing this, you conserve memory by not allowing unnecessary input values to be passed into the subclass method. You also reduce the number of variables in the workspace of your subclass methods.

Validating Inputs with Input Parser

What Is the Input Parser?

When one function calls another, passing data via an argument list to the function being called, it is the responsibility of the receiving function to verify that the incoming data meets expectations. Using the MATLAB Input Parser utility for this task ensures a consistent and thorough means of managing and validating the input information.

On calls to your function, the Input Parser:

- Validates the data passed in, throwing an exception if the validation fails.

- Assigns a default value to any inputs not passed.

- Either accepts or generates an error on any unrecognized parameter/value inputs.

- Interprets any scalar structure argument as either a single value or a series of parameter/value arguments.

- Returns information on the data passed in, arguments that defaulted, and arguments that were unrecognized, but not erroneous.

Advantages of Using the Input Parser

- Input Parser offers an established procedure to verify data input, and thus ensures consistency and thoroughness not only in your own functions, but across your design group as well.

- Using the Input Parser offers a simple interface to the user. The parsing implementation code is invisible to the user, and you can accomplish verification tasks with a minimum of easy-to-use method calls.

- You can either write your own validation routines (as full or anonymous functions) or use shared routines simply by specifying a function handle to activate them.

- The Input Parser imposes certain restrictions that apply to the passing of arguments from one function to another. These restrictions bring order to your application code which, in turn, improves its maintainability.

Working with the Example Function

The examples in this section use a function called `photoPrint` to illustrate use of the Input Parser utility. The purpose of the `photoPrint` function is to send a specified graphics file to a printer. Any function that calls `photoPrint` can pass from two to six arguments, or possibly more if they are in parameter name/value format.

As you read the documentation, look for shaded sections of program code labeled "Example Function, Part n." This code steps you through the process of creating the example function. As you encounter the shaded code segments, copy and paste them into a file called `photoPrint.m`. Then, follow the instructions for calling the function under the various conditions presented.

Beginning the Example Function

Using the MATLAB Editor, create a file named `photoPrint.m` and cut and paste the code shown here into it. The function requires that at least two input arguments be passed: `filename` and `format`. The `varargin` input allows for additional inputs.

Example Function, Part 1 — Create the Test File

Type `edit photoPrint.m` at the MATLAB command line. When the empty file appears, add the following two lines at the top:

```
function photoPrint(filename, format, varargin)
% Function for demonstrating the use of the Input Parser.
```

Calling the Example Function. The following statement shows the calling syntax for this function, including optional inputs:

```
photoPrint(filename, format, finish, ccode, 'horizDim',  dimX, ...
            'vertDim', dimY)
```

The `photoPrint` function accepts two required and up to four optional inputs. The last two inputs are in parameter name/value format and thus require two entries apiece in the argument list:

- The `filename` and `format` inputs are required

- The `finish` and `ccode` inputs are optional. Your function identifies these inputs by their position in the argument list.

- The `dimX` and `dimY` inputs are optional and are in parameter name/value format. Your function identifies these inputs by the parameter names that precede them in the argument list: 'horizDim' and 'vertDim', respectively.

The inputParser Class

The Input Parser relies on the methods and properties of the MATLAB `inputParser` class. These properties and methods provide an easy-to-use interface to all of the functionality necessary to protect your function against errors introduced by invalid data received from another function. See "Summary of inputParser Methods and Properties" on page 3-94 for a listing of all methods and properties belonging to the `inputParser` class.

Calling the constructor function for `inputParser` creates an object of the class. This object supports the creation of an input scheme representing the characteristics of each potential input argument. Construct an object `p` of the `inputParser` class in your example function.

Example Function, Part 2 — Create the Input Scheme Object

Add the following lines to `photoPrint.m` after the function definition and comment lines:

```
% Create an instance of the inputParser class.
p = inputParser;
```

Note The constructor function and all methods and property names of the `inputParser` class are case sensitive.

Validating Data Passed to a Function

There are three steps involved in preparing your function to run validation checks on the argument values passed to your function during its execution. You implement steps 2 and 3 in your function code:

Step 1 — Collect information on inputs.

Step 2 — Register expected input values with the Input Parser.

Step 3 — Parse and validate the results.

When you have completed the tasks above, your functions will perform the argument checking you have enabled on every call received from another function. The documentation that follows explains each step in detail and also implements each operation in the example function.

Step 1 — Collect Information on the Inputs to Your Function

In the planning stage of your project, collect information on all input data other functions pass in. The following table lists this information for the example function, `photoPrint`.

Input Information for Example Function. The following table shows the input scheme for the example function.

Variable Name	Type of Input	Acceptable Values	Default Value
filename	Required	Character string	N/A
format	Required	`'jpeg'`, `'tiff'`, `'png'`, `'gif'`	N/A
finish	Optional	`'flat'`, `'glossy'`, `'satin'`	`'glossy'`
colorCode	Optional	`'RGB'` or `'CMYK'`	`'CMYK'`
horizDim	Parameter/Value	Positive scalar value <= than 20	6
vertDim	Parameter/Value	Positive scalar value <= than 20	4

Writing a Validation Routine. With the Input Parser, you have the option of validating any or all of the input values passed to your function on a call. You do this by writing a validation routine for the input as part of the Input Parser code in your function. The validation routine is a function that takes one input (the value it is checking) and produces a single Boolean output. If the Boolean output is true, then MATLAB accepts the input value.

The validation routine can be a simple function handle passed directly to one of the Input Parser methods. Or it can be a more lengthy, function you have tailored for a more customized check. An example of a simple, yet useful, validation routine is one that verifies that a character string was passed:

```
@ischar
```

Another such routine is one that verifies that the value of an input lies between 0 and 100. inclusive:

```
@(x)isnumeric(x) && isscalar(x) && x>0 && x<=100);
```

Other helpful functions commonly used in argument validation are `validateattributes`, `validatestring`, and the various forms of the `is*` and `isa` functions. The documentation on "Validating the Inputs" on page 3-78 explains how the Input Parser uses the validation routine when it parses input values passed to your function.

Step 2 — Register Expected Input Values with the Input Parser

Using the information collected in step 1, register with the Input Parser the characteristics of each input value that you intend to verify whenever your

function is called. This requires calling one of three `inputParser` methods for each input.

Defining Required Inputs. Add any required arguments to the input scheme using the `addRequired` method of the class. This method takes two inputs: the name of the required parameter, and an optional handle to a function that validates the argument. Here is the syntax for the `addRequired` method:

```
p.addRequired(name, validator);
```

Add the following two `addRequired` statements to the end of your `photoPrint` code. The two arguments for `addRequired` in this case are:

- A `filename` (without its file name extension) to represent the photo file input
- A photo format keyword such as `jpeg`, `tiff`, `png`, or `gif` to identify the format of this particular file.

The validation routines check that the first input is a character string, and the second is one of four photo formats assigned to the variable `formats`.

Example Function, Part 3 — Define Required Arguments

Add the following lines to the end of `photoPrint.m`:

```
% Define inputs that one must pass on every call:
validFormats = {'jpeg', 'tiff', 'png', 'gif'};

p.addRequired('filename', @ischar);
p.addRequired('format', @(x)any(strcmp(x,validFormats)));
```

Note The order in which you invoke the `addRequired` method on different inputs determines the order in which you pass required arguments in the argument list.

Defining Optional Inputs. Use the `addOptional` method to add any optional inputs with the exception of parameter/value arguments. The syntax for `addOptional` is similar to that of `addRequired`, except that you also need to specify a default value to use whenever the optional argument is not passed:

```
addOptional(name, default, validator);
```

Add the following code segment to your `photoPrint` file. Specify a variable name, default value, and validation routine for each.

Example Function, Part 4 — Define Optional Arguments

Add the following lines to the end of `photoPrint.m`:

```
% Define inputs that default when not passed:
validFinishes = {'flat', 'glossy', 'satin'};
validCCodes = {'CMYK', 'RGB'};

p.addOptional('finish', 'glossy', ...
   @(x)any(strcmpi(x,validFinishes)));
p.addOptional('colorCode', 'CMYK', ...
   @(x)any(strcmpi(x,validCCodes)));
```

Note The order in which you invoke the `addOptional` method on different inputs determines the order in which you pass optional arguments in the argument list.

Defining Parameter/Value Inputs. Use `addParamValue` to specify those optional arguments that use a parameter/value format. The syntax is

```
addParamValue(name, default, validator);
```

Add the following code to your `photoPrint` file. You can insert the `addParamValue` commands in any order because the Input Parser recognizes them by name and not by position in the argument list. These statements define the vertical and horizontal dimensions of the photo.

Example Function, Part 5 — Define Parameter/Value Arguments

Add the following lines to the end of photoPrint.m:

```
% Define inputs passed in parameter/value format.
validateRange = @(x)validateattributes(x, {'numeric'}, ...
    {'scalar', 'integer', 'positive', '<=', 20});

p.addParamValue('horizDim', 6, validateRange);
p.addParamValue('vertDim', 4, validateRange);
```

Listing Parameters. The variable names representing the arguments in the input argument list are referred to as *parameters* of the function. To display a list of all parameters for your function, whether passed in the function call or not, examine the Parameters property of the input scheme object.

Example Function, Part 6 — Display Argument Names

Add the following lines to the end of photoPrint.m:

```
% Display the names of all arguments.
disp 'The input parameters for this program are'
disp(p.Parameters)
```

(After displaying the input parameters, comment out or remove the two disp statements shown above before going on to Part 7 of the example.)

Save the file, and then run it as shown here:

```
photoPrint('myPhoto', 'jpeg', 'satin')
```

The output is

```
The input parameters for this program are
  'colorCode' 'filename' 'finish' 'format' 'horizDim' 'vertDim'
```

Step 3 — Parse and Validate the Input Arguments

When you have an input scheme that represents all possible inputs to the function, the next step is to initiate parsing of the inputs and evaluating the results. The `parse` method of the `inputParser` class reads and validates all arguments passed by the calling function. The syntax for `parse` is

```
p.parse(arglist);
```

where `arglist` represents the arguments passed to the function.

Validating the Inputs. When your function receives data passed from another function, the Input Parser checks any arguments for which you specified a validation function. If this validator returns `false` or generates an error, MATLAB aborts the function and displays an error message.

Look back toward the start of the `photoPrint` example function and find the call made to method `addParamValue` (also shown below). This statement registers a validating function for the `horizDim` input called `validateRange`. This validator checks the value of the `horizDim` input to make sure it is a scalar, positive, integer less than or equal to 20:

```
validateRange = @(x)validateattributes(x, {'numeric'}, ...
    {'scalar', 'integer', 'positive', '<=', 20});
p.addParamValue('horizDim', 6, validateRange);
```

To see what happens when a validator fails, call `photoPrint`, this time setting `horizDim` to a number greater than 20. Input Parser responds to this by generating an error:

```
photoPrint('myPhoto', 'tiff', 'satin', 'RGB', ...
           'horizDim', 28, 'vertDim', 10)

??? Error using ==> photoPrint at 39
Argument 'horizDim' failed validation with error:
Expected input to be an array with all of the values <= 20.
```

After calling the `parse` method, your function either continues to execute using the validated input data passed to it, or the Input Parser detects an error and generates an error. This error is likely to be the result of input that did not pass validation. If your function code catches this error, you might

be able to use data available in the Results property to give a full report on what caused the error, or to correct the error and continue.

Examining One of the Values Passed. Execution of the parse method builds a structure named Results from the arguments passed from the calling function. This structure is accessible as a property of the input scheme object. To get the value of any one input (e.g., argname), type

```
p.Results.argname
```

For example, to see what value was passed for the vertDim input, examine p.Results.vertDim.

Example Function, Part 7 — Display Selected Input Values

Add the following lines to the end of photoPrint.m.

```
% Parse and validate all input arguments.
p.parse(filename, format, varargin{:});

% Show the value of a specific argument.
fprintf('\n')
fprintf('\n%s%d%s\n', ...
    'The vertical dimension of the photo is ', ...
    p.Results.vertDim, ' inches.')
```

(After displaying the values, comment out or remove the two fprintf statements shown above before going on to Part 8 of the example.)

Save and execute the file, passing at least the vertDim argument. The Input Parser assigns those values you pass to the appropriate fields of the Results structure. Display the value of the vertDim field:

```
photoPrint('myPhoto', 'tiff', 'flat', 'RGB', ...
           'horizDim', 10, 'vertDim', 8)

The vertical dimension of the photo is 8 inches.
```

Examining All Values Passed. To get the value of *all* arguments passed to photoPrint, examine the entire p.Results structure.

Example Function, Part 8 — Display All Input Values

Add the following lines to the end of `photoPrint.m`:

```
% Show the value of a specific argument.
fprintf('\n')
disp 'List of all arguments:'
disp(p.Results)
```

Save and execute the file, passing any number of the input arguments. Examining `p.Results` displays the name of each input as a field, and the value of each input as the value of that field:

```
photoPrint('myPhoto', 'tiff', 'flat', 'RGB', ...
           'horizDim', 10, 'vertDim', 8)

List of all arguments:
    colorCode: 'RGB'
     filename: 'myPhoto'
       finish: 'flat'
       format: 'tiff'
      horizDim: 10
       vertDim: 8
```

Calling a Function that Uses the Input Parser

When calling your function, put all required values first in the argument list. Follow these with all (position-based) optional values, and pass any parameter name-value arguments last. The order within each grouping is also important and is explained below.

Passing Required Inputs

When calling a function that parses its inputs with the Input Parser, pass all required inputs in the order in which they were added to the input scheme. For example, if you have a function `argTest1` with required inputs defined as follows:

```
p.addRequired('initValue', @(x)all(<x40));
```

```
p.addRequired('timeLimit', @(x)isa(x,'uint8')
p.addRequired('errors', @islogical);
```

then the first call shown below is valid, but the second, in which the order of the arguments has been altered, is not and thus generates an error:

```
argTest1(10:20, uint8(260), true)
```

```
argTest1(true, 10:25, uint8(260))
??? Error using ==> argTest1 at 7
Argument 'timeLimit' failed validation @(x)isa(x,'uint8').
```

Passing Optional (Position-Based) Inputs

As with required inputs, the order of optional inputs in an argument list must match the order in which they were added to the input scheme with the addOptional function. MATLAB identifies these inputs by their position in the argument list. For example, say that you are creating the input scheme for a function. You add four optional arguments to the scheme, the last of these being an argument named count. Whenever you call this function, you must pass the value for count fourth following the last of the *required* inputs.

```
function partSpec(height, diam, taper)
p = inputParser;
p.addRequired('height', @(x)x < 3.75
p.addOptional('diam', 11.35, @(x)x < 12);
p.addOptional('taper', 0.30, @(x)x < 0.42;
```

Normally, this is not a problem. But, consider the function shown here that has eight optional input arguments:

```
(height, width, diam, taper)
```

If you were to call this function with the intention of using default values for all inputs except for yAxis, you still need to pass the seven preceding arguments (in their default states) just to put the yAxis value in its correct position in the argument list. You can simplify this task as explained in the next section.

Passing Optional Arguments in a Nondefault Order. The `argTest2` function shown below accepts eight optional inputs. The default value and validation function for each of these inputs (e.g., `defaultVal_1` and `ValidateFun_1`) are represented by variables for the sake of simplifying the example:

```
function argTest2(varargin)
p = inputParser;
p.addOptional('arg1', defaultVal_1, validateFun_1;
p.addOptional('arg2', defaultVal_2, validateFun_2;
                    :
                    :
p.addOptional('arg7', defaultVal_7, validateFun_7;
p.addOptional('arg8', defaultVal_8, validateFun_8;);
```

When passing nondefault values for each of these inputs, you would enter a command like the following:

```
argTest2(diam, surface, bore, weight, align, taper, xAxis, yAxis)
```

But if you wanted to use mostly default values for these inputs, it seems like you would still need to pass all eith arguments in order to

wanted the function to use all default values except for just the last, you would like to be able to call it with only the one nondefault value:

```
argTest2(yAxis)
```

To do this, pass the input is as if it were a parameter name/value pair rather than a position-based argument. This requires that you have not only a *value* to pass (the value given to the `yAxis` variable, in this case), but also a *name* with which to identify the argument you are passing. You get the name from the `addOptional` statement you used when entering the input scheme for this function:

```
p.addOptional('arg8', def8, val8);
```

For the eighth input argument then, the name part of your name/value pair in this example is arg8. Now you can replace the long list of arguments shown above with the name and value of just the argument, or arguments, you want to specify nondefault values for:

```
argTest2('arg8', yAxis);
```

Passing Parameter Name/Value (Name-Based) Inputs

The number and order of parameter name/value inputs in the argument list is not important. MATLAB identifies each value using its accompanying name string argument. The only requirements for this style of input are that:

- Each name string is immediately followed in the argument list by the value it applies to.

- You do not mix optional inputs that are position-based with those that are name-based.

Even though the latter is possible in some cases, it is not recommended if you want to have easily maintainable code.

Differentiating Between the Two Types of Optional Arguments

there is insufficient information in the input scheme. A missing validation routine, for example, can make it difficult for the Input Parser software to understand how you intend to handle certain sets of inputs. Here is an example of such a situation and how to resolve it.

The following function accepts optional and parameter name/value inputs, but does not specify any validation routine for these values:

```
function argTest3(varargin)
p = inputParser;
p.addOptional('arg1', 'orange')
p.addOptional('arg2', 'green')
p.addParamValue('arg3', 'blue')
p.parse(varargin{:});
p.Results
```

Call the function with the following input values and observe that the Input Parser throws an error:

```
argTest3('str1', 'str2', 'arg3', 25)
??? Error using ==> argTest3 at 8
Argument 'str1' is a string and does not match any parameter
    names. It failed validation for 'arg1'.
```

You get the error because there are at least two ways to interpret this argument list and the Input Parser has not been given enough information to determine which was intended. Two possible interpretations are:

- The arg1 and arg2 arguments are considered to be two optional arguments. They take on the values str1 and str2, respectively. The arg3 argument and value 25 are a parameter name/value pair.

- The arg1 and arg2 arguments are considered to be the two parts of a single name/value pair. The argument is unmatched; that is, it is not specified in the input scheme. The arg3 argument and value 25 are considered a second name/value pair.

In this case, the error message indicates that MATLAB has assumed that the latter interpretation was intended. It finds that there is no parameter name in the input scheme named str1, and that is not an acceptable state.

You can alter this outcome in at least two different ways:

- You can change the setting of the KeepUnmatched property from its default state to true. This indicates to the Input Parser that two sequential, optional inputs that have no validation routine defined in the input scheme are to be interpreted as a parameter name/value pair. See "Example 1 — Keeping Unmatched Inputs" on page 3-84, below.

- You can specify a validating function in the two addOptional statements. This indicates that you have chosen to interpret two sequential, optional inputs as two distinct arguments that each must agree with the declared validator. See "Example 2 — Validating Optional Inputs" on page 3-85, below.

Example 1 — Keeping Unmatched Inputs

In the argTest3 function shown above, change the lines that follow the p.addParamValue command as shown here. This sets the KeepUnmatched property to true, and also checks whether any arguments that do not match the input scheme have been received:

```
p.KeepUnmatched = true;
p.parse(varargin{:});          % These two lines ...
results = p.Results            % were here before.
```

```
f ~isempty(fieldnames(p.Unmatched))
    fprintf('\n');
    unmatched = p.Unmatched
end
```

Making the same call as before no longer results in an error:

```
argTest3('str1', 'str2', 'arg3', 25)
results =
    arg1: 'orange'
    arg2: 'green'
    arg3: 25

unmatched =
    str1: 'str2'
```

The Input Parser accepts the input string `'arg3'` and the value 25 that follows it as a name/value pair. It also accepts string inputs `str1` and `str2` as the two parts of a second, but unmatched, parameter name/value pair. Because all inputs are now accounted for, the Input Parser assigns to the remaining arguments, `arg1` and `arg2`, their default values.

Example 2 — Validating Optional Inputs

Leave the `KeepUnmatched` property in its default state (`false`), and modify the two calls to `addOptional` in the `argTest3` file by adding a validating function that expects a character string:

```
p.addOptional('arg1', 'orange', @ischar)
p.addOptional('arg2', 'green', @ischar)
```

Now call the function as you did in the previous example:

```
argTest3('str1', 'str2', 'arg3', 25)
results =
    arg1: 'str1'
    arg2: 'str2'
    arg3: 25
```

The validation routines for the first two arguments now make it clear that two separate, optional inputs are expected in the argument list.

Substituting Default Values for Arguments Not Passed

When the caller of your function passes fewer than the full number of inputs in the argument list, the Input Parser substitutes default values for those arguments that were not specified. This assumes, of course, that the unspecified arguments were all defined as optional and given default values when creating the input scheme.

If requested, the Input Parser returns a cell array `p.UsingDefaults` that lists any inputs that were not passed in the argument list, and thus were replaced with default values. If none of the inputs defaulted, then `p.UsingDefaults` is an empty cell array.

Example Function, Part 9 — Show Defaulted Inputs

Add the following lines to the end of `photoPrint.m`:

```
% Show any arguments not specified in the call.
defaulted = p.UsingDefaults;
if ~isempty(defaulted)
    fprintf('\n')
    disp 'List of arguments given default values:'
    disp(defaulted)
end
```

Save the file and run it without specifying the `colorCode` or `vertDim` arguments:

```
photoPrint('myPhoto', 'tiff', 'flat', 'horizDim', 8)
```

At the end of the output, you should see

```
List of arguments given default values:
    'colorCode'
    'vertDim'
```

Handling Unmatched Inputs

The Input Parser generates an error if your function is called with any arguments that are not part of its input scheme. You can disable this error by setting the `KeepUnmatched` property to `true`. When `KeepUnmatched` is in the `true` state, the Input Parser does not throw an error, but instead stores any arguments that are not in the input scheme in a structure accessible through the `Unmatched` property of the object.

The `KeepUnmatched` property defaults to `false`. If all inputs match the scheme, then `p.Unmatched` is a 1-by-1 struct array with no fields.

Note Whenever you change the value of any writeable property of `inputParser`, make sure that you make this change above the line containing the call to the `parse` method. Otherwise, it has no effect on the parse.

Example Function, Part 10 — Show Unmatched Inputs

Insert the following statement into your example code, making sure that it precedes the `p.parse` statement:

```
p.KeepUnmatched = true;
```

Then add the following lines at the end of the function code:

```
% List the names of any arguments not listed in the input scheme.
newArgs = p.Unmatched;   fCell = fieldnames(newArgs);
if ~isempty(fCell)
    fprintf('\n')
    disp 'List of unmatched arguments:'
    disp(newArgs)
end
```

Save and run the function, passing two arguments that are not defined in the scheme.

```
photoPrint('myPhoto', 'tiff', 'satin', 'RGB', 'count', 7, ...
```

```
                    'Name', 'Paul James')
```

At the end of the output, you should see

```
List of unmatched arguments:
    count: 7
     Name: Paul James'
```

Interpreting Arguments Passed as Structures

When the Input Parser receives an argument that is a scalar structure, it can interpret the argument as either a single value that is to be assigned to a variable, or a set of optional parameter/value arguments corresponding to the structure's field names and values. Setting an optional switch in the Input Parser determines how a structure argument is to be interpreted.

Passing Arguments Packaged In a Structure

By setting the StructExpand property of the inputParser object to true, you can pass optional arguments to your function in the form of a structure instead of individually in the argument list. Your function code must set StructExpand prior to calling the parse method.

Continue with the photoPrint example from the previous section.

Example Function, Part 11 — Expanding a Structure Input

Insert the following statement into your example code, making sure that it precedes the p.parse statement:

```
p.StructExpand = true;
```

At the MATLAB command line, put the optional input arguments into a structure, s:

```
s.finish = 'flat';      s.colorCode = 'RGB';
s.vertDim = 16;         s.horizDim = 10;
```

Now call the function, passing all of the required arguments followed by the structure argument, s:

```
photoPrint('myPhoto', 'tiff', s)
```

Input Parser displays the results:

```
List of all arguments:
    colorCode: 'RGB'
     filename: 'myPhoto'
       finish: 'flat'
       format: 'tiff'
      horizDim: 10
      vertDim: 16
```

Note If input structure s contains any fields that do not match variable names in the input scheme, the Input Parser generates an error unless the keepUnmatched property is set to true.

Overriding Arguments Passed In a Structure

If you want to pass your argument list in a structure, as described in the previous section, but you also want to alter the value of one or more of these arguments without having to modify the structure, you can do so by passing both the structure and the modified argument.

This part of the example passes the value for horizDim in two places in the argument list. Its value is 10 in s.horizDim, and 20 in the parameter/value argument. The separate argument value overrides the structure field value:

```
s.finish = 'flat';        s.colorCode = 'RGB';
s.vertDim = 16;           s.horizDim = 10;

photoPrint('myPhoto', 'tiff', s, 'horizDim', 20);
List of all arguments:
    colorCode: 'RGB'
     filename: 'myPhoto'
       finish: 'flat'
       format: 'tiff
      horizDim: 20
      vertDim: 16
```

Passing Data Packaged In a Structure

To pass a structure that you want to keep together as a single argument value, rather than expand into multiple values, set the StructExpand property to false.

Example Function, Part 12 — Receiving a Scalar Struct Input

Go back up to where you set p.StructExpand to true and change its value to false:

```
p.StructExpand = false;
```

Using the same structure s as in the previous example,

```
s.finish = 'flat';        s.colorCode = 'RGB';
s.vertDim = 16;           s.horizDim = 10;
```

call the example function passing 'testStruct' and s as a parameter/value pair:

```
photoPrint('myPhoto', 'tiff', 'flat', 'RGB', 'testStruct', s);
```

You can see the effect of turning off the StructExpand option. In this case, the Input Parser has not expanded the structure input s. Instead, it is the value of a new input called testStruct which is shown in the "unmatched arguments" list:

```
List of all arguments:
    colorCode: 'RGB'
     filename: 'myPhoto'
       finish: 'flat'
       format: 'tiff'
      horizDim: 6
       vertDim: 4

List of arguments given default values:
    'horizDim'    'vertDim'

List of unmatched arguments:
    testStruct: [1x1 struct]
```

Other Features of the Input Parser

The Input Parser also makes it easy to:

- Enable or disable case-sensitive matching.
- Specify the function name to be used in the error message.
- Make copies of the input scheme for your function.

Enabling Case-Sensitive Matching

When you pass optional arguments in the function call, the Input Parser compares the names of these arguments with the argument names in the input scheme. By default, this comparison is not sensitive to letter case. An argument name defined in the scheme as `vertDim` matches an argument passed as `VERTDIM` in the function call.

You can override the default and make these comparisons case sensitive by setting the `CaseSensitive` property of the object to `true`. MATLAB does not error on a case mismatch, unless the `KeepUnmatched` property is set to `false` (its default state).

Set `KeepUnmatched` to `false` and `CaseSensitive` to `true` in your `photoPrint` file.

Example Function, Part 13 — Making Validation Case Sensitive

Go back up in your example code to the statement that set the `KeepUnmatched` property. Change the value of `KeepUnmatched` to `false`, and also set the `CaseSensitive` property to `true`:

```
p.KeepUnmatched = false;
p.CaseSensitive = true;
```

Save and run the function, using `'VERTDIM'` as the name of the argument for specifying the vertical dimension of the photo to be printed:

```
photoPrint('myPhoto', 'tiff', 'satin', 'RGB', ...
```

```
                    'horizDim', 8, 'VERTDIM', 10)

??? Error using ==> photoPrint at 42
Argument 'VERTDIM' did not match any valid parameter of
  the parser.
```

Case-Sensitive Matching with KeepUnmatched Enabled. This behavior changes when you set `KeepUnmatched` back to `true`. In this state, the argument that does not match due to its letter case is considered to be an additional mismatched argument rather than an error.

Example Function, Part 14 — Case Sensitive Matching with No Error

Go back up in the example code to the statement that assigns the `KeepUnmatched` property. Change its value to `true`:

```
  p.KeepUnmatched = true;
```

Run the following command and observe that the Input Parser displays the `VERTDIM` variable and its value in its list of unmatched arguments. MATLAB does not generate an error in this case:

```
photoPrint('myPhoto', 'tiff', 'satin', 'RGB', ...
           'horizDim', 8, 'VERTDIM', 10)

List of all arguments:
    colorCode: 'RGB'
     filename: 'myPhoto'
       finish: 'satin'
       format: 'tiff'
      horizDim: 8
      vertDim: 4

List of arguments given default values:
    'vertDim'

List of unmatched arguments:
    VERTDIM: 10
```

Adding the Function Name to Error Messages

Use the FunctionName property to include a function name of your choosing in error messages thrown by the Input Parser. This applies only to errors thrown by those validating functions you defined with the addRequired, addOptional, or addParamValue methods.

Example Function, Part 15 — Identifying the Function on an Error

Assign a name to the FunctionName property. (This would more commonly be set to the name of the currently running function. The example sets it to a different value to make its effect more noticeable.) Again, make sure that this assignment is done prior to executing the parse command.

```
p.FunctionName = 'My test photoPrint function';
```

Save and run the function and observe text of the error message:

```
photoPrint('myPhoto', 'tiff', 'eggshell')

?? Error using ==> My test photoPrint function
Argument 'finish' failed validation @(x)any(strcmp(x,finishes)).
```

Making a Copy of the Input Scheme

The createCopy method enables you to make a copy of an existing input scheme. Because the inputParser class uses handle semantics, you cannot make a copy of the object using an assignment statement.

The following statement creates an inputParser object s that is a copy of p:

```
s = p.createCopy
```

Summary of inputParser Methods and Properties

Methods and Properties Used in Preparing the Input Scheme

Method	Description
addRequired	Define a required argument.
addOptional	Define an optional argument.
addParamValue	Define a parameter/value argument.
createCopy	Create a copy of the inputParser object

Property	Description	Data Structure	Access
Parameters	Names of arguments defined in the input scheme.	1xN cell array of strings	Read

Methods and Properties Used in Parsing Inputs to Your Function

Method	Description
parse	Parse and validate the named inputs.

Property	Description	Data Structure	Access
Results	Names and values of arguments passed in a function call that are in the input scheme.	1x1 structure	Read

Properties Used in Evaluating the Results

Property	Description	Data Structure	Default	Access
KeepUnmatched	Enable or disable errors on unmatched arguments.	1x1 logical	false	Write
StructExpand	Enable or disable passing arguments in a structure.	1x1 logical	true	Write
CaseSensitive	Enable or disable case-sensitive matching of argument names.	1x1 logical	false	Write
FunctionName	Function name to be included in error messages.	1x1 logical	empty string	Write
Unmatched	Names and values of arguments passed in function call that are not in the input scheme for this function.	1x1 structure	N/A	Read
UsingDefaults	Names of arguments not passed in function call that are given default values.	1xN cell array	N/A	Read

Functions Provided By MATLAB

Overview

Many of the functions provided with the MATLAB software are implemented as program files just like the files you create with MATLAB. Other MATLAB functions are precompiled executable programs called *built-ins* that run much more efficiently. Many MATLAB functions are also overloaded so that they handle different classes appropriately.

Functions

If you look in the subfolders of the toolbox\matlab folder, you can find the sources to many of the functions supplied with MATLAB. Locate your toolbox\matlab folder by typing

```
dir([matlabroot '\toolbox\matlab\'])
```

Any MATLAB functions that you write are just like any other functions coded with MATLAB. When one of these functions is called, MATLAB parses and executes each line of code in the file. It saves the parsed version of the function in memory, eliminating parsing time on any further calls to this function.

Identifying Functions

To find out if a function is implemented with a program file, use the exist function. The exist function searches for the name you enter on the MATLAB path and returns a number identifying the source. If the source is a file with a .m file extension, then exist returns the number 2. This example identifies the source for the repmat function as a program file:

```
exist repmat
ans =
     2
```

The `exist` function also returns 2 for files that have a file type unknown to MATLAB. However, if you invoke `exist` on a MATLAB function name, the file type is known to MATLAB and returns 2 only on program files.

Viewing the Source Code

One advantage of functions implemented as files is that you can look at the source code. This can help when you need to understand why the function returns a value you did not expect, if you need to figure out how to code something in MATLAB that is already coded in a function, or perhaps to help you create a function that overloads one of the MATLAB functions.

To find the source code for any MATLAB function, use `which`:

```
which repmat
   D:\matlabR14\toolbox\matlab\elmat\repmat.m
```

Built-In Functions

Functions that are frequently used or that can take more time to execute are often implemented as executable files. These functions are called *built-ins*.

Unlike MATLAB program file functions, you cannot see the source code for built-ins. Although most built-in functions do have a program file associated with them, this file is there mainly to supply the help documentation for the function. Take the `reshape` function, for example, and find it on the MATLAB path:

```
which reshape
   D:\matlabR14\toolbox\matlab\elmat\reshape.m
```

If you type this file out, you will see that it consists almost entirely of help text. At the bottom is a call to the built-in executable image.

Identifying Built-In Functions

As with program file functions, you can identify which functions are built-ins using the `exist` function. This function identifies built-ins by returning the number 5:

```
exist reshape
ans =
     5
```

Forcing a Built-In Call

If you overload any of the MATLAB built-in functions to handle a specific class, then MATLAB always calls the overloaded function on that type. If, for some reason, you need to call the built-in version, you can override the usual calling mechanism using the `builtin` function. The expression

```
builtin('reshape', arg1, arg2, ..., argN);
```

forces a call to the MATLAB built-in function, `reshape`, passing the arguments shown even though an overload exists for the class in this argument list.

Note With the exception of overloading, you should not create a MATLAB program file that has the same name as a MATLAB built-in. Because built-in functions have a higher precedence than most other types of program files (with the exception of private and subfunctions), MATLAB does not recognize functions that share the same name with a built-in.

Overloaded MATLAB Functions

An *overloaded function* is an additional implementation of an existing function that is designed specifically to handle a certain class. When you pass an argument of this type in a call to the function, MATLAB looks for the function implementation that handles that type and executes that function code.

Each overloaded MATLAB function has a file on the MATLAB path. The files for a certain class reside in a folder named with an @ sign followed by the class name. For example, if you need to plot expressions of class `polynom` in a manner that is unique to that class, you can overload the MATLAB `plot`

function. To do this, create your own plotting function `plot.m` specifically for use with objects of the polynom class. Then, create a folder called `@polynom`, and store your own version of `plot.m` in that folder.

You can add your own overloads to any function. Just create a class folder for the class you want to support for that function, and create a file that handles the type in a manner different from the default. See Defining Classes — Syntax and Developing Classes — Typical Workflow.

When you use the `which` command with the `-all` option, MATLAB returns all occurrences of the file you are looking for. This is an easy way to find functions that are overloaded:

```
which -all set          % Show all implementations for 'set'
```

Internal Utility Functions

MathWorks reserves the use of packages named `internal` for utility functions used by internal MATLAB code. Functions that belong to an `internal` package are intended for MathWorks use only. Using functions that belong to an `internal` package is strongly discouraged. These functions are not guaranteed to work in a consistent manner from one release to the next. In fact, any of these functions and classes could be removed from the MATLAB software in any subsequent release without notice and without documentation in the product release notes.

Any function called with a syntax that begins with the package name `internal` is an internal function. For example,

```
internal.matlab.functionname
```

Any function on the MATLAB path that resides at any level under a folder named `+internal` is an internal function. For example,

```
matlab\toolbox\matlab\+internal\functionname
```

4

Types of Functions

- "Overview of MATLAB Function Types" on page 4-2
- "Anonymous Functions" on page 4-3
- "Primary Functions" on page 4-15
- "Nested Functions" on page 4-16
- "Subfunctions" on page 4-33
- "Private Functions" on page 4-35
- "Overloaded Functions" on page 4-37

Overview of MATLAB Function Types

There are essentially two ways to create a new function for your MATLAB application: in a command entered at run-time, or in a file saved to permanent storage.

The command-oriented function, called an *anonymous function*, is relatively brief in its content. It consists of a single MATLAB statement that can interact with multiple input and output arguments. The benefit of using anonymous functions is that you do not have to edit and maintain a file for functions that require only a brief definition.

There are several types of functions that you write and execute as a file. The most basic of these are *primary functions* and *subfunctions*. Primary functions are visible to other functions outside of the file they are defined in, while subfunctions, generally speaking, are not. That is, you can call a primary function from an anonymous function or from a function defined in a separate file, but you can call a subfunction only from functions within the same file. (See the Description section of the `function_handle` reference page for information on making a subfunction externally visible.)

Two specific types of primary functions are the *private* and *overloaded function*. Private functions are visible only to a limited group of other functions. This type of function can be useful if you want to limit access to a function, or when you choose not to expose the implementation of a function. Overloaded functions act the same way as overloaded functions in most computer languages. You can create multiple implementations of a function so that each responds accordingly to different types of inputs.

The last type of MATLAB function is the *nested function*. Nested functions are not an independent function type; they exist within the body of one of the other types of functions discussed here (with the exception of anonymous functions), and also within other nested functions.

Anonymous Functions

Constructing an Anonymous Function

Anonymous functions give you a quick means of creating simple functions without having to store your function to a file each time. You can construct an anonymous function either at the MATLAB command line or in any function or script.

The syntax for creating an anonymous function from an expression is

```
fhandle = @(arglist) expr
```

Starting from the right of this syntax statement, the term `expr` represents the body of the function: the code that performs the main task your function is to accomplish. This consists of any single, valid MATLAB expression. Next is `arglist`, which is a comma-separated list of input arguments to be passed to the function. These two components are similar to the body and argument list components of any function.

Leading off the entire right side of this statement is an `@` sign. The `@` sign is the MATLAB operator that constructs a function handle. Creating a function handle for an anonymous function gives you a means of invoking the function. It is also useful when you want to pass your anonymous function in a call to some other function. The `@` sign is a required part of an anonymous function definition.

> **Note** Function handles not only provide access to anonymous functions. You can create a function handle to any MATLAB function. The constructor uses a different syntax: fhandle = @functionname (e.g., fhandle = @sin). To find out more about function handles, see "Function Handles" on page 1-127 in the Programming Fundamentals documentation.

The syntax statement shown above constructs the anonymous function, returns a handle to this function, and stores the value of the handle in variable fhandle. You can use this function handle in the same way as any other MATLAB function handle.

Simple Example

The statement below creates an anonymous function that finds the square of a number. When you call this function, MATLAB assigns the value you pass in to variable x, and then uses x in the equation x.^2:

```
sqr = @(x) x.^2;
```

The @ operator constructs a function handle for this function, and assigns the handle to the output variable sqr. As with any function handle, you execute the function associated with it by specifying the variable that contains the handle, followed by a comma-separated argument list in parentheses. The syntax is

```
fhandle(arg1, arg2, ..., argN)
```

To execute the sqr function defined above, type

```
a = sqr(5)
a =
   25
```

Because sqr is a function handle, you can pass it in an argument list to other functions. The code shown here passes the sqr anonymous function to the MATLAB quad function to compute its integral from zero to one:

```
quad(sqr, 0, 1)
ans =
```

```
    0.3333
```

A Two-Input Example

As another example, you could create the following anonymous function that uses two input arguments, x and y. Variables A and B are already defined:

```
A = [2 3 4];    B = [5 6 7];
sumAxBy = @(x, y) (A*x + B*y);

whos sumAxBy
Name          Size                    Bytes  Class

sumAxBy      1x1                        16  function_handle
```

To call this function, assigning 5 to x and 7 to y, type

```
sumAxBy(5, 7)
```

Evaluating With No Input Arguments

For anonymous functions that do not take any input arguments, construct the function using empty parentheses for the input argument list:

```
t = @() datestr(now);
```

Also use empty parentheses when invoking the function:

```
t()

ans =
04-Sep-2003 10:17:59
```

You must include the parentheses. If you type the function handle name with no parentheses, MATLAB just identifies the handle; it does not execute the related function:

```
t

t =
    @() datestr(now)
```

Arrays of Anonymous Functions

To store multiple anonymous functions in an array, use a cell array. The example shown here stores three simple anonymous functions in cell array A:

```
A = {@(x)x.^2, @(y)y+10, @(x,y)x.^2+y+10}
A =
    @(x)x.^2     @(y)y+10     @(x,y)x.^2+y+10
```

Execute the first two functions in the cell array by referring to them with the usual cell array syntax, A{1} and A{2}:

```
A{1}(4) + A{2}(7)
ans =
    33
```

Do the same with the third anonymous function that takes two input arguments:

```
A{3}(4, 7)
ans =
    33
```

Space Characters in Anonymous Function Elements

Note that while using space characters in the definition of any function can make your code easier to read, spaces in the body of an anonymous function that is defined in a cell array can sometimes be ambiguous to MATLAB. To ensure accurate interpretation of anonymous functions in cell arrays, you can do any of the following:

- Remove all spaces from at least the body (not necessarily the argument list) of each anonymous function:

  ```
  A = {@(x)x.^2, @(y)y+10, @(x, y)x.^2+y+10};
  ```

- Enclose in parentheses any anonymous functions that include spaces:

  ```
  A = {(@(x)x .^ 2), (@(y) y +10), (@(x, y) x.^2 + y+10)};
  ```

- Assign each anonymous function to a variable, and use these variable names in creating the cell array:

```
A1 = @(x)x .^ 2;  A2 = @(y) y +10;  A3 = @(x, y)x.^2 + y+10;
A = {A1, A2, A3};
```

Outputs from Anonymous Functions

As with other MATLAB functions, the number of outputs returned by an anonymous function depends mainly on how many variables you specify to the left of the equals (=) sign when you call the function.

For example, consider an anonymous function getPersInfo that returns a person's address, home phone, business phone, and date of birth, in that order. To get someone's address, you can call the function specifying just one output:

```
address = getPersInfo(name);
```

To get more information, specify more outputs:

```
[address, homePhone, busPhone] = getPersInfo(name);
```

Of course, you cannot specify more outputs than the maximum number generated by the function, which is four in this case.

Example

The anonymous getXLSData function shown here calls the MATLAB xlsread function with a preset spreadsheet filename (records.xls) and a variable worksheet name (worksheet):

```
getXLSData = @(worksheet) xlsread('records.xls', worksheet);
```

The records.xls worksheet used in this example contains both numeric and text data. The numeric data is taken from instrument readings, and the text data describes the category that each numeric reading belongs to.

Because the MATLAB xlsread function is defined to return up to three values (numeric, text, and raw data), getXLSData can also return this same number of values, depending on how many output variables you specify to the left of the equals sign in the call. Call getXLSData a first time, specifying only a single (numeric) output, dNum:

```
dNum = getXLSData('Week 12');
```

Display the data that is returned using a `for` loop. You have to use generic names (v1, v2, v3) for the categories due to the fact that the text of the real category names was not returned in the call:

```
for k = 1:length(dNum)
  disp(sprintf('%s    v1: %2.2f    v2: %d    v3: %d', ...
      datestr(clock, 'HH:MM'), dNum(k,1), dNum(k,2), ...
      dNum(k,3)));
  end
```

Here is the output from the first call:

```
12:55    v1: 78.42    v2: 32    v3: 37
13:41    v1: 69.73    v2: 27    v3: 30
14:26    v1: 77.65    v2: 17    v3: 16
15:10    v1: 68.19    v2: 22    v3: 35
```

Now try this again, but this time specifying two outputs, numeric (dNum) and text (dTxt):

```
[dNum, dTxt] = getXLSData('Week 12');

for k = 1:length(dNum)
  disp(sprintf('%s    %s: %2.2f    %s: %d    %s: %d', ...
      datestr(clock, 'HH:MM'), dTxt{1}, dNum(k,1), ...
      dTxt{2}, dNum(k,2), dTxt{3}, dNum(k,3)));
  end
```

This time, you can display the category names returned from the spreadsheet:

```
12:55    Temp: 78.42    HeatIndex: 32    WindChill: 37
13:41    Temp: 69.73    HeatIndex: 27    WindChill: 30
14:26    Temp: 77.65    HeatIndex: 17    WindChill: 16
15:10    Temp: 68.19    HeatIndex: 22    WindChill: 35
```

Variables Used in the Expression

Anonymous functions commonly include two types of variables:

- Variables specified in the argument list. These often vary with each function call.

- Variables specified in the body of the expression. MATLAB captures these variables and holds them constant throughout the lifetime of the function handle.

The latter variables must have a value assigned to them at the time you construct an anonymous function that uses them. Upon construction, MATLAB captures the current value for each variable specified in the body of that function. The function will continue to associate this value with the variable even if the value should change in the workspace or go out of scope.

The fact that MATLAB captures the values of these variables when the handle to the anonymous function is constructed enables you to execute an anonymous function from anywhere in the MATLAB environment, even outside the scope in which its variables were originally defined. But it also means that to supply new values for any variables specified within the expression, you must reconstruct the function handle.

Changing Variables Used in an Anonymous Function

The second statement shown below constructs a function handle for an anonymous function called `parabola` that uses variables `a`, `b`, and `c` in the expression. Passing the function handle to the MATLAB `fplot` function plots it out using the initial values for these variables:

```
a = 1.3;   b = .2;   c = 30;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```
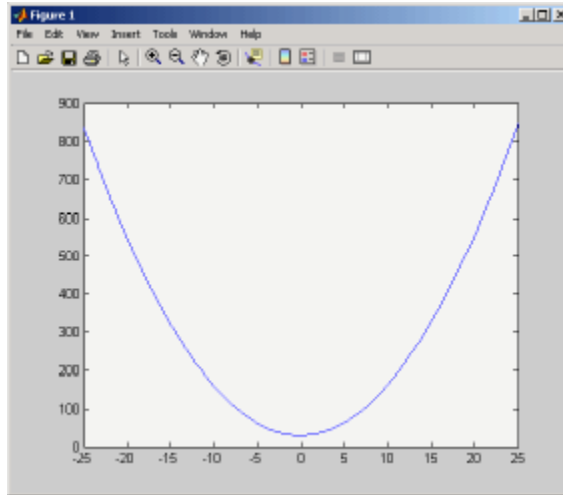
If you change the three variables in the workspace and replot the figure, the parabola remains unchanged because the parabola function is still using the initial values of a, b, and c:

```
a = -3.9;    b = 52;    c = 0;
fplot(parabola, [-25 25])
```
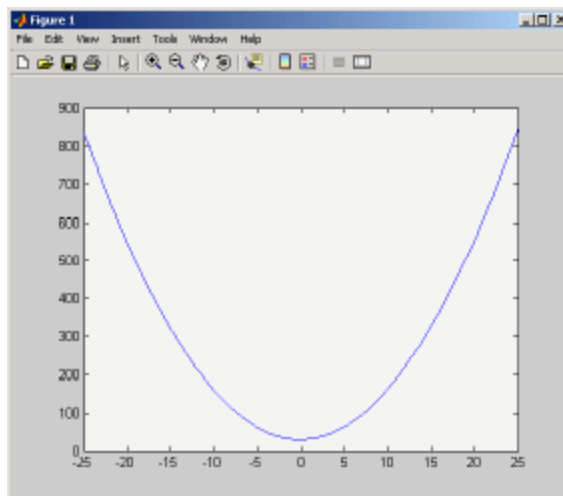
To get the function to use the new values, you need to reconstruct the function handle, causing MATLAB to capture the updated variables. Replot using the new construct, and this time the parabola takes on the new values:

```
a = -3.9;   b = 52;   c = 0;
parabola = @(x) a*x.^2 + b*x + c;
fplot(parabola, [-25 25])
```
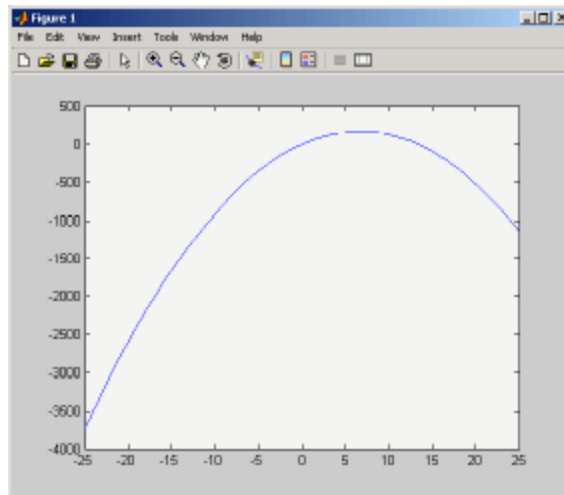

For the purposes of this example, there is no need to store the handle to the anonymous function in a variable (parabola, in this case). You can just construct and pass the handle right within the call to fplot. In this way, you update the values of a, b, and c on each call:

```
fplot(@(x) a*x.^2 + b*x + c, [-25 25])
```

Examples of Anonymous Functions

This section shows a few examples of how you can use anonymous functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

Example 1 — Passing a Function to quad

The equation shown here has one variable t that can vary each time you call the function, and two additional variables, g and omega. Leaving these two variables flexible allows you to avoid having to hardcode values for them in the function definition:

```
x = g * cos(omega * t)
```

One way to program this equation is to write a function, and then create a function handle for it so that you can pass the function to other functions, such as the MATLAB quad function as shown here. However, this requires creating and maintaining a new file for a purpose that is likely to be temporary, using a more complex calling syntax when calling quad, and passing the g and omega parameters on every call. Here is the function:

```
function f = vOut(t, g, omega)
f = g * cos(omega * t);
```

This code has to specify g and omega on each call:

```
g = 2.5;   omega = 10;

quad(@vOut, 0, 7, [], [], g, omega)
ans =
   0.1935

quad(@vOut, -5, 5, [], [], g, omega)
ans =
   -0.1312
```

You can simplify this procedure by setting the values for g and omega just once at the start, constructing a function handle to an anonymous function that only lasts the duration of your MATLAB session, and using a simpler syntax when calling quad:

```
g = 2.5;   omega = 10;
f = @(t) (g * cos(omega * t));
```

```
 quad(f, 0, 7)
ans =
   0.1935

quad(f, -5, 5)
ans =
  -0.1312
```

To preserve an anonymous function from one MATLAB session to the next, save the function handle to a MAT-file

```
save anon.mat f
```

and then load it into the MATLAB workspace in a later session:

```
load anon.mat f
```

Example 2 — Multiple Anonymous Functions

This example solves the following equation by combining two anonymous functions:

$$g(c) = \int_{0}^{1} (x^2 + cx + 1)dx$$

The equivalent anonymous function for this expression is

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));
```

This was derived as follows. Take the parenthesized part of the equation (the integrand) and write it as an anonymous function. You do not need to assign the output to a variable as it will only be passed as input to the quad function:

```
@(x) (x.^2 + c*x + 1)
```

Next, evaluate this function from zero to one by passing the function handle, shown here as the entire anonymous function, to quad. You need to temporarily set c to some value to test this:

```
c = 2;

quad(@(x) (x.^2 + c*x + 1), 0, 1)
ans =
    2.3333
```

Supply the value for c by constructing an anonymous function for the entire equation and you are done:

```
g = @(c) (quad(@(x) (x.^2 + c*x + 1), 0, 1));

g(2)
ans =
    2.3333
```

Primary Functions

The first function in any MATLAB program file is called the *primary function*. Following the primary function can be any number of subfunctions, which can serve as subroutines to the primary function.

Under most circumstances, the primary function is the only function in the file that you can call from the MATLAB command line or from another function. You invoke this function using the name of the file in which it is defined.

For example, the `average` function shown here resides in the file `average.m`:

```
function y = average(x)
% AVERAGE Mean of vector elements.

y = sum(x)/length(x);       % Actual computation
```

You can invoke this function from the MATLAB command line with this command to find the average of three numbers:

```
average([12 60 42])
```

Note that it is customary to give the primary function the same name as the file in which it resides. If the function name differs from the filename, then you must use the filename to invoke the function.

Nested Functions

Writing Nested Functions

You can define one or more functions within another function in your MATLAB application. These inner functions are said to be *nested* within the function that contains them. You can also nest functions within other nested functions. You cannot however define a nested function inside any of the MATLAB program control statements. This includes any block of code that is controlled by an `if`, `else`, `elseif`, `switch`, `for`, `while`, `try`, or `catch` statement.

To write a nested function, simply define one function within the body of another function in your program. Like any function, a nested function contains any or all of the components described in "Basic Parts of a Program File" on page 3-10 in the Programming Fundamentals documentation. In addition, you must always terminate a nested function with an `end` statement:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end
...
end
```

Note Functions do not normally require a terminating end statement. This rule does not hold, however, when you nest functions. If a program file contains one or more nested functions, you must terminate *all* functions (including subfunctions) in the file with end, whether or not they contain nested functions.

Example — More Than One Nested Function

This example shows function A and two additional functions nested inside A at the same level:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
   end

   function z = C(p4)
   ...
   end
...
end
```

Example — Multiply Nested Functions

This example shows multiply nested functions, C nested inside B, and B in A:

```
function x = A(p1, p2)
...
   function y = B(p3)
   ...
      function z = C(p4)
      ...
      end
   ...
   end
...
end
```

Calling Nested Functions

You can call a nested function

- From the level immediately above it. (In the following code, function A can call B or D, but not C or E.)

- From a function nested at the same level within the same parent function. (Function B can call D, and D can call B.)

- From a function at any lower level. (Function C can call B or D, but not E.)

```
function A(x, y)            % Primary function
B(x, y);
D(y);

   function B(x, y)         % Nested in A
   C(x);
   D(y);

      function C(x)         % Nested in B
      D(x);
      end
   end

   function D(x)            % Nested in A
   E(x);

      function E(x)         % Nested in D
      ...
      end
   end
end
```

You can also call a subfunction from any nested function in the same file.

You can pass variable numbers of arguments to and from nested functions, but you should be aware of how MATLAB interprets varargin, varargout, nargin, and nargout under those circumstances. See "Passing Optional Arguments to Nested Functions" in the MATLAB Programming Fundamentals documentation for more information on this.

> **Note** If you construct a function handle for a nested function, you can call the nested function from any MATLAB function that has access to the handle. See "Using Function Handles with Nested Functions" on page 4-21.

Nested functions are not accessible to the `str2func` or `feval` function. You cannot call a nested function using a handle that has been constructed with `str2func`. And, you cannot call a nested function by evaluating the function name with `feval`. To call a nested function, you must either call it directly by name, or construct a function handle for it using the `@` operator.

Variable Scope in Nested Functions

The scope of a variable is the range of functions that have direct access to the variable to set, modify, or acquire its value. When you define a local (i.e., nonglobal) variable within a function, its scope is normally restricted to that function alone. For example, subfunctions do not share variables with the primary function or with other subfunctions. This is because each function and subfunction stores its variables in its own separate workspace.

Like other functions, a nested function has its own workspace. But it also has access to the workspaces of all functions in which it is nested. So, for example, a variable that has a value assigned to it by the primary function can be read or overwritten by a function nested at any level within the primary. Similarly, a variable that is assigned in a nested function can be read or overwritten by any of the functions containing that function.

In the following two examples, variable `x` is stored in the workspace of the outer `varScope` function and can be read or written to by all functions nested within it.

```
function varScope1              function varScope2
x = 5;                          nestfun1
nestfun1                           function nestfun1
   function nestfun1                  nestfun2
      nestfun2
                                      function nestfun2
      function nestfun2                  x = 5;
         x = x + 1                    end
      end                          end
   end                          x = x + 1
end                             end
```

As a rule, a variable used or defined within a nested function resides in the
workspace of the outermost function that both contains the nested function
and accesses that variable. The scope of this variable is then the function to
which this workspace belongs, and all functions nested to any level within
that function.

In the next example, the outer function, varScope3, does not access variable
x. Following the rule just stated, x is unknown to the outer function and
thus is not shared between the two nested functions. In fact, there are
two separate x variables in this example: one in the function workspace of
nestfun1 and one in the function workspace of nestfun2. When nestfun2
attempts to update x, it fails because x does not yet exist in this workspace:

```
function varScope3
nestfun1
nestfun2

   function nestfun1
      x = 5;
   end

   function nestfun2
      x = x + 1
   end
end
```

The Scope of Output Variables

Variables containing values returned by a nested function are not in the scope of outer functions. In the two examples shown here, the one on the left fails in the second to last line because, although the *value* of y is returned by the nested function, the *variable* y is local to the nested function, and unknown to the outer function. The example on the right assigns the return value to a variable, z, and then displays the value of z correctly.

Incorrect	Correct
| ```
function varScope4
x = 5; nestfun;
   function y = nestfun
      y = x + 1;
   end
y
end
``` | ```
function varScope5
x = 5;
z = nestfun;
   function y = nestfun
       y = x + 1;
   end
z
end
``` |

Using Function Handles with Nested Functions

Every function has a certain *scope*, that is, a certain range of other functions to which it is visible. A function's scope determines which other functions can call it. You can call a function that is out of scope by providing an alternative means of access to it in the form of a function handle. (The function handle, however, must be within the scope of its related function when you construct the handle.) Any function that has access to a function handle can call the function with which the handle is associated.

Note Although you can call an out of scope function by means of a function handle, the handle itself must be within the scope of its related function at the time it is constructed.

The section on "Calling Nested Functions" on page 4-18 defines the scope of a nested function. As with other types of functions, you can make a nested function visible beyond its normal scope with a function handle. The following function getCubeHandle constructs a handle for nested function findCube

and returns its handle, h, to the caller. The @ sign placed before a function name (e.g., @findCube) is the MATLAB operator that constructs a handle for that function:

```
function h = getCubeHandle
h = @findCube;                    % Function handle constructor

    function cube = findCube(X)   % Nested function
        cube = X .^ 3;
    end
end
```

Call getCubeHandle to obtain the function handle to the nested function findCube. Assign the function handle value returned by getCubeHandle to an output variable, cubeIt in this case:

```
cubeIt = getCubeHandle;
```

You can now use this variable as a means of calling findCube from outside of its program file:

```
cubeIt(8)
ans =
    512
```

Note When calling a function by means of its handle, use the same syntax as if you were calling a function directly. But instead of calling the function by its name (e.g., strcmp(S1, S2)), use the variable that holds the function handle (e.g., fhandle(S1, S2)).

Function Handles and Nested Function Variables

One characteristic of nested functions that makes them different from other MATLAB functions is that they can share nonglobal variables with certain other functions in the same file. A nested function nFun can share variables with any outer function that contains nFun, and with any function nested within nFun. This characteristic has an impact on how certain variables are stored when you construct a handle for a nested function.

Defining Variables When Calling Via Function Handle. The example
below shows a primary function getHandle that returns a function handle for
the nested function nestFun. The nestFun function uses three different types
of variables. The VLoc variable is local to the nested function, VInp is passed in
when the nested function is called, and VExt is defined by the outer function:

```
function h = getHandle(X)
h = @nestFun;
VExt = someFun(X);

    function nestFun(VInp)
    VLoc = 173.5;
    doSomeTask(VInp, VLoc, VExt);
    end
end
```

As with any function, when you call nestFun, you must ensure that you
supply the values for any variables it uses. This is a straightforward matter
when calling the nested function directly (that is, calling it from getHandle).
VLoc has a value assigned to it within nestFun, VInp has its value passed in,
and VExt acquires its value from the workspace it shares with getHandle.

However, when you call nestFun using a function handle, only the nested
function executes; the outer function, getHandle, does not. It might seem at
first that the variable VExt, otherwise given a value by getHandle, has no
value assigned to it in the case. What in fact happens though is that MATLAB
stores variables such as VExt inside the function handle itself when it is being
constructed. These variables are available for as long as the handle exists.

The VExt variable in this example is considered to be *externally scoped* with
respect to the nested function. Externally scoped variables that are used in
nested functions for which a function handle exists are stored within the
function handle. So, function handles not only contain information about
accessing a function. For nested functions, a function handle also stores the
values of any externally scoped variables required to execute the function.

Example Using Externally Scoped Variables

The sCountFun and nCountFun functions shown below return function handles
for subfunction subCount and nested function nestCount, respectively.

These two inner functions store a persistent value in memory (the value is retained in memory between function calls), and then increment this value on every subsequent call. subCount makes its count value persistent with an explicit persistent declaration. In nestCount, the count variable is externally scoped and thus is maintained in the function handle:

Using a Subfunction	Using a Nested Function
| ```
function h = sCountFun(X)
h = @subCount;
count = X
subCount(0, count);
function subCount(incr, ini)
persistent count;
initializing = nargin > 1;
if initializing
   count = ini; else
   count = count + incr
end
``` | ```
function h = nCountFun(X)
h = @nestCount;
count = X
   function nestCount(incr)
      count = count + incr
   end
end
``` |

When sCountFun executes, it passes the initial value for count to the subCount subfunction. Keep in mind that the count variable in sCountFun is not the same as the count variable in subCount; they are entirely independent of each other. Whenever subCount is called via its function handle, the value for count comes from its persistent place in memory.

In nestCount, the count variable again gets its value from the primary function when called from within the file. However, in this case the count variable in the primary and nested functions are one and the same. When nestCount is called by means of its function handle, the value for count is assigned from its storage within the function handle.

Running the Example. The subCount and nestCount functions increment a value in memory by another value that you pass as an input argument. Both of these functions give the same results.

Get the function handle to nestCount, and initialize the count value to a four-element vector:

```
h = nCountFun([100 200 300 400])
```

```
count =
   100   200   300   400
```

Increment the persistent vector by 25, and then by 42:

```
h(25)
count =
   125   225   325   425

h(42)
count =
   167   267   367   467
```

Now do the same using sCountFun and subCount, and verify that the results are the same.

Note If you construct a new function handle to subCount or nestCount, the former value for count is no longer retained in memory. It is replaced by the new value.

Separate Instances of Externally Scoped Variables

The code shown below constructs two separate function handles to the same nested function, nestCount, that was used in the last example. It assigns the handles to fields counter1 and counter2 of structure s. These handles reference different instances of the nestCount function. Each handle also maintains its own separate value for the externally scoped count variable.

Call nCountFun twice to get two separate function handles to nestCount. Initialize the two instances of count to two different vectors:

```
s.counter1 = nCountFun([100 200 300 400]);
count =
   100   200   300   400

s.counter2 = nCountFun([-100 -200 -300 -400]);
count =
  -100  -200  -300  -400
```

Now call `nestCount` by means of each function handle to demonstrate that MATLAB increments the two `count` variables individually.

Increment the first counter:

```
s.counter1(25)
count =
   125   225   325   425
s.counter1(25)
count =
   150   250   350   450
```

Now increment the second counter:

```
s.counter2(25)
count =
   -75  -175  -275  -375
s.counter2(25)
count =
   -50  -150  -250  -350
```

Go back to the first counter and you can see that it keeps its own value for `count`:

```
s.counter1(25)
count =
   175   275   375   475
```

Restrictions on Assigning to Variables

The scoping rules for nested, and in some cases anonymous, functions require that all variables used within the function be present in the text of the code. Adding variables to the workspace of this type of function at run time is not allowed.

MATLAB issues an error if you attempt to dynamically add a variable to the workspace of an anonymous function, a nested function, or a function that contains a nested function. Examples of operations that might use dynamic assignment in this way are shown in the table below.

Type of Operation	How to Avoid Using Dynamic Assignment
Evaluating an expression using `eval` or `evalin`, or assigning a variable with `assignin`	As a general suggestion, it is best to avoid using the `eval`, `evalin`, and `assignin` functions altogether.
Loading variables from a MAT-file with the `load` function	Use the form of `load` that returns a MATLAB structure.
Assigning to a variable in a MATLAB script	Convert the script to a function, where argument- and result-passing can often clarify the code as well.
Assigning to a variable in the MATLAB debugger	You can declare the variable to be `global`. For example, to create a variable X for temporary use in debugging, use `K>> global X;    X = value`

One way to avoid this error in the other cases is to pre-declare the variable in the desired function.

Examples of Nested Functions

This section shows a few examples of how you can use nested functions. These examples are intended to show you how to program with this type of function. For more mathematically oriented examples, see the MATLAB Mathematics documentation.

The examples in this section include

- "Example 1 — Creating a Function Handle for a Nested Function" on page 4-27
- "Example 2 — Function-Generating Functions" on page 4-29

Example 1 — Creating a Function Handle for a Nested Function

The following example constructs a function handle for a nested function and then passes the handle to the MATLAB `fplot` function to plot the parabola

shape. The makeParabola function shown here constructs and returns a function handle fhandle for the nested parabola function. This handle gets passed to fplot:

```
function fhandle = makeParabola(a, b, c)
% MAKEPARABOLA returns a function handle with parabola
% coefficients.

fhandle = @parabola;     % @ is the function handle constructor

    function y = parabola(x)
    y = a*x.^2 + b*x + c;
    end
end
```

Assign the function handle returned from the call to a variable (h) and evaluate the function at points 0 and 25:

```
h = makeParabola(1.3, .2, 30)
h =
    @makeParabola/parabola

h(0)
ans =
    30

h(25)
ans =
  847.5000
```

Now pass the function handle h to the fplot function, evaluating the parabolic equation from x = -25 to x = +25:

```
fplot(h, [-25 25])
```


Example 2 — Function-Generating Functions

The fact that a function handle separately maintains a unique instance of the function from which it is constructed means that you can generate multiple handles for a function, each operating independently from the others. The function in this example makes IIR filtering functions by constructing function handles from nested functions. Each of these handles maintains its own internal state independent of the others.

The function makeFilter takes IIR filter coefficient vectors a and b and returns a filtering function in the form of a function handle. Each time a new input value x_n is available, you can call the filtering function to get the new output value y_n. Each filtering function created by makeFilter keeps its own private a and b vectors, in addition to its own private state vector, in the form of a transposed direct form II delay line:

```
function [filtfcn, statefcn] = makeFilter(b, a)
%   FILTFCN = MAKEFILTER(B, A) creates an IIR filtering
%   function and returns it in the form of a function handle,
```

```
%    FILTFCN. Each time you call FILTFCN with a new filter
%    input value, it computes the corresponding new filter
%    output value, updating its internal state vector at the
%    same time.
%
%    [FILTFCN, STATEFCN] = MAKEFILTER(B, A) also returns a
%    function (in the form of a function handle, STATEFCN)
%    that can return the filter's internal state.  The internal
%    state vector is in the form of a transposed direct form
%    II delay line.

%    Initialize state vector. To keep this example a bit
%    simpler, assume that a and b have the same length.
%    Also assume that a(1) is 1.

v = zeros(size(a));

filtfcn =  @iirFilter;
statefcn = @getState;

   function yn = iirFilter(xn)
      % Update the state vector
      v(1) = v(2) + b(1) * xn;
      v(2:end-1) = v(3:end) + b(2:end-1) * xn - ...
         a(2:end-1) * v(1);
      v(end) = b(end) * xn - a(end) * v(1);

      % Output is the first element of the state vector.
      yn = v(1);
   end

   function vOut = getState
      vOut = v;
   end
end
```

This sample session shows how makeFilter works. Make a filter that has
a decaying exponential impulse response and then call it a few times in
succession to see the output values change:

```
[filt1, state1] = makeFilter([1 0], [1 -.5]);

% First input to the filter is 1.
filt1(1)
ans =
     1

% Second input to the filter is 0.
filt1(0)
ans =
    0.5000

filt1(0)
ans =
    0.2500

% Show the filter's internal state.
state1()
ans =
    0.2500    0.1250

% Hit the filter with another impulse.
filt1(1)
ans =
    1.1250

% How did the state change?
state1()
ans =
    1.1250    0.5625

% Make an averaging filter.
filt2 = makeFilter([1 1 1]/3, [1 0 0]);

% Put a step input into filt2.
filt2(1)
ans =
    0.3333

filt2(1)
```

```
ans =
    0.6667

filt2(1)
ans =
    1

% The two filter functions can be used independently.
filt1(0)
ans =
    0.5625
```

As an extension of this example, suppose you were looking for a way to develop simulations of different filtering structures and compare them. This might be useful if you were interested in obtaining the range of values taken on by elements of the state vector, and how those values compare with a different filter structure. Here is one way you could capture the filter state at each step and save it for later analysis:

Call `makeFilter` with inputs `v1` and `v2` to construct function handles to the `iirFilter` and `getState` subfunctions:

```
[filtfcn, statefcn] = makeFilter(v1, v2);
```

Call the `iirFilter` and `getState` functions by means of their handles, passing in random values:

```
x = rand(1, 20);
for k = 1:20
    y(k) = filtfcn(x(k));
    states{k} = statefcn();  % Save the state at each step.
end
```

Subfunctions

Overview

MATLAB program files can contain code for more than one function. Additional functions within the file are called *subfunctions*, and these are only visible to the primary function or to other subfunctions in the same file.

Each subfunction begins with its own function definition line. The functions immediately follow each other. The various subfunctions can occur in any order, as long as the primary function appears first:

```
function [avg, med] = newstats(u) % Primary function
% NEWSTATS Find mean and median with internal functions.
n = length(u);
avg = mean(u, n);
med = median(u, n);

function a = mean(v, n)          % Subfunction
% Calculate average.
a = sum(v)/n;

function m = median(v, n)        % Subfunction
% Calculate median.
w = sort(v);
if rem(n, 2) == 1
   m = w((n+1) / 2);
else
   m = (w(n/2) + w(n/2+1)) / 2;
end
```

The subfunctions `mean` and `median` calculate the average and median of the input list. The primary function `newstats` determines the length of the list and calls the subfunctions, passing to them the list length `n`.

Subfunctions cannot access variables used by other subfunctions, even within the same file, or variables used by the primary function of that file, unless you declare them as global within the pertinent functions, or pass them as arguments.

Calling Subfunctions

When you call a function from within a program file, MATLAB first checks the file to see if the function is a subfunction. It then checks for a private function (described in the following section) with that name, and then for a standard program file or built-in function on your search path. Because it checks for a subfunction first, you can override existing files using subfunctions with the same name.

Accessing Help for a Subfunction

You can write help for subfunctions using the same rules that apply to primary functions. To display the help for a subfunction, precede the subfunction name with the name of the file that contains the subfunction (minus file extension) and a > character.

For example, to get help on subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

Private Functions

In this section...
"Overview" on page 4-35
"Private Folders" on page 4-35
"Accessing Help for a Private Function" on page 4-36

Overview

Private functions are functions that reside in subfolders with the special name `private.` These functions are called *private* because they are visible only to functions and scripts that meet these conditions:

- A function that calls a private function must be defined in a program file that resides in the folder immediately above that `private` subfolder.

- A script that calls a private function must itself be called from a function that has access to the private function according to the above rule.

For example, assume the folder `newmath` is on the MATLAB search path. A subfolder of `newmath` called `private` can contain functions that only the functions in `newmath` can call.

Because private functions are invisible outside the parent folder, they can use the same names as functions in other folders. This is useful if you want to create your own version of a particular function while retaining the original in another folder. Because MATLAB looks for private functions before standard functions, it finds a private function named `test.m` before a nonprivate program file named `test.m`.

Primary functions and subfunctions can also be implemented as private functions.

Private Folders

You can create your own private folders simply by creating subfolders called `private` using the standard procedures for creating folders on your computer. Do not place these private folders on your path.

Accessing Help for a Private Function

You can write help for private functions using the same rules that apply to primary functions. To display the help for a private function, precede the private function name with `private/`.

For example, to get help on private function `myprivfun`, type

```
help private/myprivfun
```

Overloaded Functions

Overloaded functions are useful when you need to create a function that responds to different types of inputs accordingly. For instance, you might want one of your functions to accept both double-precision and integer input, but to handle each type somewhat differently. You can make this difference invisible to the user by creating two separate functions having the same name, and designating one to handle `double` types and one to handle integers.

MATLAB overloaded functions reside in subfolders having a name starting with the symbol @ and followed by the name of a recognized MATLAB class. For example, functions in the `\@double` folder execute when invoked with arguments of MATLAB type `double`. Those in an `\@int32` folder execute when invoked with arguments of MATLAB type `int32`.

See "Overloading MATLAB Functions" for more information on overloading functions in MATLAB.

5

Using Objects

MATLAB Objects

In this section...
"Getting Oriented" on page 5-2
"Getting Comfortable with Objects" on page 5-2
"What Are Objects and Why Use Them?" on page 5-2
"Accessing Objects" on page 5-3
"Objects In the MATLAB Language" on page 5-4
"Other Kinds of Objects Used by MATLAB" on page 5-4

Getting Oriented

This chapter provides information for people using objects. It does not provide a thorough treatment of object-oriented concepts, but instead focuses on what you need to know to use the objects provided with MATLAB.

If you are interested in object-oriented programming in the MATLAB language, see *Object-Oriented Programming*. For background information on objects, see object-oriented design.

Getting Comfortable with Objects

MATLAB uses objects because they are a convenient way to package data. Working with objects in MATLAB is like working with any variables and is often more convenient because objects are optimized for specific purposes. Think of an object as a neatly packaged collection of data that includes functions that operate on the data. The documentation for any particular object describes how to use it.

What Are Objects and Why Use Them?

In the simplest sense, objects are special-purpose variables that have a specific set of operations that you can perform on the data they contain. You do not need to know how the operations are implemented or how the data is stored. This makes objects modular and easy to pass within application

programs. It also isolates your code from changes to the object's design and implementation.

In a more general sense, objects are organized collections of data and functions that have been designed for specific purposes. For example, an object might be designed to contain time series data that consists of value/time-sample pairs and associated information like units, sample uniformity, and so on. This object could have a set of specific operations designed to perform analysis, such as filtering, interpolating, and plotting. The following sections provide examples of such objects.

Accessing Objects

You access an object with its variable name. Interacting with objects variables in MATLAB software is really no different from interacting with any other variables. Basically, you can perform the same common operations on variables whether they hold numbers or specialized objects. For example, you can do the following things with objects:

- Create it and assigned a variable name so you can reference it again

- Assign or reassign data to it (see "Accessing Object Data" on page 5-14)

- Operate on its data (see "Calling Object Methods" on page 5-16)

- Convert it to another class (if this operation is supported by the object's class)

- Save it to a MAT-file so you can reload it later (see `save`)

- Copy it (see "Copying Objects" on page 5-26)

- Clear it from the workspace (`clear`)

Any particular object might have restrictions on how you create it, access its data, or what operations you can perform on it. Refer to the documentation for the particular MATLAB object for a description of what you can do with that object.

See "Variables" on page 2-8 for a general discussion of MATLAB variables.

Objects In the MATLAB Language

The MATLAB language uses many specialized objects. For example,
MException objects capture information when errors occur, timer objects
execute code at a certain time interval, the serial object enables you to
communicate with devices connected to your computer's serial port, and so
on. MATLAB toolboxes often define objects to manage the specific data and
analyses performed by the toolbox.

All of these objects are designed to provide specific functionality that is not as
conveniently available from general purpose language components.

Other Kinds of Objects Used by MATLAB

The MATLAB language enables you to use other kinds of objects in your
MATLAB programs. The following objects are different from the MATLAB
objects described in this documentation. See the individual sections
referenced below for information on using these objects.

- Handle Graphics® objects represent objects used to create graphs and
  GUIs. These objects provide a set/get interface to property values, but
  are not extensible by subclassing. See "Handle Graphics Objects" for more
  information.

- Sun Java objects can be used in MATLAB code enabling you to access the
  capabilities of Java classes. See "Using Sun Java Classes in MATLAB
  Software" for more information.

- Microsoft COM objects enable you to integrate these software components
  into your application. See "COM Support for MATLAB Software" for more
  information.

- Microsoft .NET objects enable you to integrate .NET assemblies into
  your application. See "MATLAB Interface to .NET Framework" for more
  information.

- User-defined MATLAB objects created prior to Version 7.6 used different
  syntax for class definition (no classdef block) and exhibit other differences.
  See "Compatibility with Previous Versions " for more information.

General Purpose Vs. Specialized Arrays

In this section...
"How They Differ" on page 5-5
"Using General-Purpose Data Structures" on page 5-5
"Using Specialized Objects" on page 5-6

How They Differ

The MATLAB language enables you to use both general-purpose and specialized arrays. For example, numeric multidimensional arrays, `struct`, and `cell` arrays provide general-purpose data storage. You typically extract data from the array and pass this data to functions (e.g., to perform mathematical analysis). Then, you store the data back in general-purpose arrays.

When using a specialized object, you typically pass the object's data to a function that creates the object. Once you have created the object, you use specially defined functions to operate on the data. These functions are unique to the object and are designed specifically for the type and structure of the data contained by the object.

Using General-Purpose Data Structures

A commonly used general-purpose data structure references data via fieldnames. For example, these statements create a MATLAB `struct` (a MATLAB structure array):

```
s.Data = rand(10,1);
s.Time = .01:.01:.1;
s.Name = 'Data1';
s.Units = 'seconds';
```

The structure `s` contains two arrays of numbers. However, `s` is a generic type in the sense that MATLAB does not define special functions to operate on the data in this particular structure. For example, while `s` contains two fields that would be useful to plot, `Data` and `Time`, you cannot pass `s` to the `plot` function:

```
plot(s)
??? Error using ==> plot
Not enough input arguments.
```

While s has enough information to create a plot of `Data` versus `Time`, `plot` cannot access this data because structures like s can contain any values in its fields and the fields can have any name. Just because one field is named `Data` does not force you to assign data to that field.

To plot the data in s, you have to extract the data from the fields, pass them as numeric arrays in the desired order to the `plot` function, add a title, labels, and so on:

```
plot(s.Time,s.Data)
title(['Time Series Plot: ' s.Name])
xlabel(['Time (' s.Units ')'])
ylabel(s.Name)
```

You could create a function to perform these steps for you. Other programs using the structure s would need to create their own functions or access the one you created.

Using Specialized Objects

Compare the array s above to an object that you have designed specifically to contain and manipulate time series data. For example, the following statement creates a MATLAB `timeseries` object. It is initialized to store the same data as the structure s above:

```
tsobj = timeseries(rand(10,1),.01:.01:.1,'Name','Data1');
```

The function that creates the object `tsobj`, accepts sample data, sample times, a property name/property value pair (`Name`/`Data1`), and uses a default value of `Units` (which is `seconds`).

The designer of this object created a special version of the `plot` function that works directly with this object. For example:

```
plot(tsobj)
```

Notice how the object's `plot` function creates a graph that is plotted and labeled with the data from the `tsobj` object. As a user of this object, you do not need write your own code to produce this graph. The class design specifies the standard way to present graphs of `timeseries` data and all clients of this object use the same code for plotting.

See "Time Series Objects" for more on using MATLAB `timeseries` objects.

Key Object Concepts

In this section...
"Basic Concepts" on page 5-8
"Classes Describe How to Create Objects" on page 5-8
"Properties Contain Data" on page 5-9
"Methods Implement Operations" on page 5-9
"Events are Notices Broadcast to Listening Objects" on page 5-10

Basic Concepts

There are some basic concepts that are fundamental to objects. Objects represent something in the real world, like an error condition or the set of data you collected in a product test. Objects enable you to do something useful, like provide an error report or analyze and present the results of tests.

There are basic components that MATLAB uses to realize the design of an object. These components include:

- Classes
- Properties
- Methods
- Events

Classes Describe How to Create Objects

A *class* defines a set of similar objects. It is a description from which MATLAB creates a particular instance of the class, and it is the instance (that is, the object) that contains actual data. Therefore, while there is a `timeseries` class, you work with `timeseries` objects.

Classes are defined in code files — either as separate `.m` files or built-in to the MATLAB executable. Objects are specific representations of a class that you access through workspace variables.

Properties Contain Data

Objects store data in *properties*. Consider a `timeseries` object as an example. `timeseries` object properties contains time series data, corresponding time values, and related information, such as units, events, data quality, and interpolation method. MATLAB objects enable you to access property data directly (see "Accessing Object Data" on page 5-14 for information on property syntax).

Properties are sometimes called fields in other programming languages and are similar to the fields of MATLAB structures. Properties have descriptive names, such as `Data` and `DataInfo`, in the case of `timeseries` objects, and can contain any kind of MATLAB data, including other objects.

An object, then, is a container for a predefined set of data. Unlike a cell array or structure, you cannot add new properties or delete defined properties from an object. Doing so would compromise the object's intended purpose and violate the class design.

The class design can restrict the values you can assign to a property. For example, a `Length` property might restrict possible values to positive integers or might be read only and determine its own value when queried.

Methods Implement Operations

Class *methods* are functions designed to work with objects of a particular class. Methods enable the class designer to implement specific operations that are optimized for the data contained in the object. You do not have to extract the data from the object, modify its format, and pass it to a general-purpose MATLAB function because the class defines methods with an awareness of the object's structure.

Methods can define operations that are unique to a particular class of object, such as adding a data sample to an existing set of time series data, or can *overload* common operations in a way that makes sense for the particular object. For example, `timeseries` objects have an `addsample` method to add a new data sample to an existing `timeseries` object. Also, `timeseries` overloads the MATLAB `plot` function to work with `timeseries` objects.

MATLAB software determines which overloaded version of a method to call based on the class of the object passed as an argument. If you execute a MATLAB statement like:

```
tsobjnew = tsobj1 + tsobj2;
```

where `tsobj1` and `tsobj2` are `timeseries` objects, MATLAB calls the `timeseries` version of the + operation (if defined) and returns a new `timeseries` object.

Because the `timeseries` class defines the operation, you can add a `timeseries` object to a scalar number:

```
tsobjnew = tsobj1 + 4;
```

The class definition determines what happens when you add a scalar `double` to a `timeseries` object (the scalar is added to each `Data` value).

Methods make working with objects convenient for the user, but also provide advantages to the class designer. Methods hide implementation details from users—you do not need to create your own functions to access and manipulate data, as you would when using general-purpose data structures like `structs` and cell arrays. This provides the flexibility to change the internal design of an object without affecting object clients (i.e., application programs that use the objects).

Events are Notices Broadcast to Listening Objects

Classes can defined names for specific actions and trigger the broadcast of notices when those actions occur. Listeners respond to the broadcast of an event notice by executing a predefined function.

For example, objects can listen for the change of the value of a property and execute a function when that change occurs. If an object defines an event for which you can define a listening object, the object's documentation describes that event. See "Events — Sending and Responding to Messages" for information on how class designers use events.

Creating Objects

In this section...
"Class Constructor" on page 5-11
"When to Use Package Names" on page 5-11

Class Constructor

Usually, you create an object by calling a function designed for the purpose of creating that specific class of object. For example, the following code creates a timeseries object and assigns it to the variable tsboj:

```
load count.dat % Load some data
tsobj = timeseries(count(:,1),1:24,'Name','Data1');
```

The timeseries constructor creates an object and initializes its data with the values specified as arguments. Classes that create objects define a special method whose purpose is to create objects of the class. This method has the same name as the class and is called the *class constructor*.

However, in some cases, you might create objects by calling other functions or even using a GUI. For example, a try-catch block can return an MException object that contains information about a specific error condition. In this case, you do not explicitly create the object, rather it is returned by the catch statement (see "Accessing Object Data" on page 5-14 for an example).

When to Use Package Names

A package is a container that provides a logical grouping for class and function definitions. The class and function names within a given package must be unique, but can be reused in other packages. Packages are folders that begin with the + character.

If a package folder contains a class definition, then you must use the package name when calling the class constructor. For example, this statement creates a Map object, whose class definition file is in a folder in the containers package:

```
mapobj = containers.Map({'rose','bicycle'},{'flower','machine'});
```

You need to use the package name to refer to:

- Class constructors (e.g., `containers.Map`), which you call to create an object

- Static methods (methods that do not require an object of the class as an argument)

- Package functions (functions defined in the package)

However, because MATLAB uses the class of an object to determine which ordinary method to call, you do not need to use the package name in conjunction with object references. For example, suppose you have the following folder structure:

```
pathfolder/+packagename/@ClassName/ClassName.m
pathfolder/+packagename/@ClassName/staticMethodName.m
pathfolder/+packagename/functionName.m
```

In the following examples, `obj` is the object you are creating.

```
% Create object of ClassName
obj = packagename.ClassName(...);

% Call methodName
obj.methodName(...);

% Set or get the value of property PropertyName
obj.PropertyName = x;
x = obj.PropertyName;

% Call static method staticMethodName
packagename.ClassName.staticMethodName(...);

% Call package function functionName
packagename.functionName(...)
```

If a package folder contains a class definition file, then consider the package name as part of the class name. Wherever you need to use the class name, include the package name. For example, `containers.Map` is the full class name of the `Map` class.

See the object's user documentation for the syntax you need to use to create objects.

See "Organizing Classes in Folders" and "Scoping Classes with Packages" for more information on the use of packages.

See "Importing Classes" for information on importing packages into functions.

Accessing Object Data

In this section...
"Listing Public Properties" on page 5-14
"Getting Property Values" on page 5-14
"Setting Property Values" on page 5-15

Listing Public Properties

Note You should always treat property names as being case sensitive.

You can see the names of all public object properties using the properties function with the object's class name or with an actual object. For example:

```
>> properties('MException')
Properties for class MException:
    identifier
    message
    cause
    stack
```

Getting Property Values

After creating an object, you can access the values of its properties:

```
try
   a = rand(4);
   a(17) = 7;
catch me % catch creates an MException object named me
   disp(['Current error identifier: ' me.identifier])
end
Current error identifier: MATLAB:indexed_matrix_cannot_be_resized
```

Access the data in properties using dot notation:

```
object.PropertyName
```

For example, you can access the `message` property of the `MException` object, `me`, with this syntax:

```
>> me.message
ans =
 In an assignment  A(I) = B, a matrix A cannot be resized.
```

See "Capturing Information About the Error" on page 6-5 for more information on using `MException` objects.

Setting Property Values

Objects often restrict what values you can assign to them. For example, the following `timeseries` object has 10 data values, each corresponding to a sample time:

```
tsobj = timeseries(rand(10,1),1:10,'Name','Random Sample');
```

Now suppose you attempt to set the `Data` property to a three-element vector:

```
>> tsobj.Data = [1 2 3];
??? Error using ==> timeseries.timeseries>timeseries.set.Data at 171
Size of the data array is incompatible with the time vector.
```

The `timeseries` class design ensures that the number of data samples matches the number of time samples. This illustrates one of the advantages a specialized object has over a general purpose-data structure like a MATLAB `struct`.

Calling Object Methods

In this section...
"What Operations Can You Perform" on page 5-16
"Method Syntax" on page 5-16
"Class of Objects Returned by Methods" on page 5-18

What Operations Can You Perform

Methods define all aspects of an object's behavior. Consequently, most classes implement many methods that an object user is unlikely to call directly. The user documentation for the object you are using describes the operations you can perform on any particular object.

You can list the methods defined by a class with the methods or methodsview functions:

```
methods('timeseries')

Methods for class timeseries:
addevent              gettsbetweenevents    set
addsample             horzcat               setabstime
createTstoolNode      idealfilter           setinterpmethod
ctranspose           init                  setprop
...
gettsatevent          pvset                 var
gettsbeforeatevent    rdivide               vertcat
gettsbeforeevent      resample

Static methods:
tsChkTime             tsgetrelativetime
```

Method Syntax

You call an object's method using dot notation:

```
returnedValue = object.MethodName(args,...)
```

You also can call a method using function syntax, passing the object as the first (left-most) argument.

For example, `MException` objects have a `getReport` method that returns information about the error.

```
try
    surf
catch me
    disp(me.getReport)
end

Error using ==> surf at 50
Not enough input arguments.
```

Dot and function notation are usually equivalent. That is, both of the following statements return the `MException` report:

```
rpt = getReport(me); % Call getReport using function notation
rpt = me.getReport;  % Call getReport using dot notation
```

Calling the Correct Method

It is possible for the function syntax to call an unexpected method if there is more than one object in the argument list. Suppose there are two classes, `ClassA` and `ClassB`, that define a method called `addData`. Suppose further that `ClassA` is defined as being inferior to `ClassB` in precedence (something that the class designer can do in the class definition). In this situation, given `objA` is of `ClassA` and `objB` is of `ClassB`, the following two statement call different methods:

```
addData(objA,objB) % Calls objB.addData
objA.addData(objB) % Calls objA.addData
```

If `ClassA` and `ClassB` had equal precedence, then the left-most argument determines which method MATLAB calls (i.e., `objA.addData` in both statements).

It is unlikely that you will encounter this particular scenario, however, if you are calling a method that accepts more than one object as arguments, using

dot notation removes any ambiguity about which object's method MATLAB calls.

Class of Objects Returned by Methods

While methods sometimes return objects of the same class, this is not always the case. For example, the MException object's getReport returns a character string:

```
try
   surf
catch me
   rpt = me.getReport;
end

whos
  Name        Size              Bytes  Class          Attributes
  me          1x1                 780  MException
  rpt         1x171               342  char
```

Methods can return any type of value and properties can contain any type of value. However, class constructor methods always return an object or array of objects of the same type as the class.

Desktop Tools Are Object Aware

In this section...

Tab Completion Works with Objects

MATLAB tab completion works with objects. For example, if you enter an object name followed by a dot:

```
tsobj.
```

and then press the tab key, MATLAB pops up a selection box with a list of properties and methods:

The more letters you complete after the dot, the more specific is the list. See "Completing Statements in the Command Window — Tab Completion" for more information.

Editing Objects with the Variable Editor

You can use the MATLAB Variable Editor to edit object properties. To open an object in the Variable Editor, you can double-click the object name in the Workspace browser or use the openvar command:

```
tsobj = timeseries(rand(10,1),.01:.01:.1,'Name','Data1');
openvar tsobj
```

See "Viewing and Editing Workspace Variables with the Variable Editor" for more information.

Getting Information About Objects

In this section...
"The Class of Workspace Variables" on page 5-21
"Information About Class Members" on page 5-23
"Logical Tests for Objects" on page 5-23
"Displaying Objects" on page 5-24
"Getting Help for MATLAB Objects" on page 5-25

The Class of Workspace Variables

Knowing the class of the variables you are working with enables you to use them most effectively. For example, consider the following variable created in your workspace:

```
load count.dat % Load some data
tsobj = timeseries(count(:,1),1:24,'Name','Data1');
>> whos
  Name        Size              Bytes  Class
  count       24x3                576  double
  tsobj       24x1                261  timeseries
```

The whos command lists information about your workspace variables. Notice that the variable loaded from the count.dat file (count) is an array of doubles. You know, therefore, that you can perform indexing and arithmetic operations on this array. For example:

```
newcount = sum(count,2);
newcount(8:15) = NaN;
bar(newcount)
```

Indexed assignment and the bar function work with inputs of class double.

However, the `timeseries` class does not define a `bar` method for `timeseries` objects. The `timeseries` class defines a `plot` method for graphing because the class design specified a line plot as the best way to represent time series data.

Extracting Data From Object Properties

Suppose you have a `timeseries` object and you want to work directly with the numeric values of the `timeseries` data. You can extract data from the object properties and assign these values to an array. For example

```
load count.dat
tsobj = timeseries(sum(count,2),1:24,'Name','DataSum');
d = tsobj.Data;
t = tsobj.Time;
n = tsobj.Name;
d(8:15) = NaN;
bar(t,d); title(n)
```

Testing for the Class of an Object

Suppose you create a function that operates on more than one class of object. If you have a `timeseries` object, you call the `timeseries` `plot` method, but

if the object is of class double, you can call the bar function (which isn't supported by timeseries objects). You could use isa as in the following code to make this determination:

```
if isa(obj,'timeseries')
    plot(obj)
elseif isa(obj,'double')
    bar(obj)
end
```

Information About Class Members

These functions provide information about the object.

Function	Purpose
class	Return class of object
events	List of event names defined by the class
methods	List of methods implemented by the class
methodsview	Information on class methods in separate window
properties	List of class property names

Logical Tests for Objects

In functions, you might need conditional statements to determine the status of an object before performing certain actions. For example, you might perform different actions based on the class of an object (see "Testing for the Class of an Object" on page 5-22). The following functions provide logical tests for objects:

Function	Purpose
isa	Determine whether argument belongs to a particular class. True for object's class and all of object's superclasses.
isequal	Determine if two objects are equal.
isobject	Determine whether the input is a MATLAB object.

Testing for Object Equality

isequal finds two objects to be equal if all the following conditions are met:

- Both objects are of the same class

- Both objects are of the same size

- All corresponding property values are equal

isequal tests the value of every array element in every property and every property of every object contained in the objects being tested. As contained objects are tested for equality, MATLAB calls each object's own version of isequal (if such versions exist).

If objects contain large amounts of data stored in other objects, then testing for equality can be a time-consuming process.

Identifying MATLAB Objects

The isobject function returns true only for MATLAB objects. For Sun Java objects, use isjava. For Handle Graphics objects, use ishandle.

Note ishandle returns false for MATLAB handle objects. See "Testing for Handle or Value Class" on page 5-30 for more information.

Displaying Objects

When you issue commands that return objects and do not terminate those commands with a semicolon, or when you pass an object to the disp function, MATLAB displays information about the object. For example:

```
hobj = containers.Map({'Red Sox','Yankees'},
{'Boston','New York'})
hobj =
  containers.Map handle
  Package: containers
  Properties:
        Count: 2
      KeyType: 'char'
    ValueType: 'char'
```

Methods, Events, Superclasses

This information includes links (shown in blue) to documentation on the object's class and superclasses, and lists of methods, events, and superclasses. Properties and their current values are also listed.

Some classes (timeseries, for example) redefine how they display objects to provide more useful information for this particular class.

Getting Help for MATLAB Objects

You can get documentation for MATLAB objects using the doc command with the class name. To see the reference pages for the objects used in this chapter, use the following commands:

```
doc timeseries
doc MException
doc containers.Map % Include the package name
```

Copying Objects

In this section...
"Two Copy Behaviors" on page 5-26
"Value Object Copy Behavior" on page 5-26
"Handle Object Copy Behavior" on page 5-27
"Testing for Handle or Value Class" on page 5-30

Two Copy Behaviors

There are two fundamental kinds of MATLAB classes—handles and values.

Value classes create objects that behave like ordinary MATLAB variables with respect to copy operations. Copies are independent values. Operations that you perform on one object do not affect copies of that object.

Handle classes create objects that are sometimes referred to as references. This is because a handle, and all copies of this handle, refer to the same underlying object. When you create a handle object, you can copy the handle, but not the data referenced by the object's properties. Any operations you perform on a handle object affects all copies of that object. Handle Graphics objects behave in this way.

Value Object Copy Behavior

MATLAB numeric variables exhibit the behavior of value objects. For example, when you copy a to the variable b, both variables are independent of each other. Changing the value of a does not change the value of b:

```
a = 8;
b = a;
```

Now reassign a and b is unchanged:

```
a = 6;
b
b =
     8
```

Clearing a does not affect b:

```
clear a
b
b =
     8
```

Value Object Properties

The copy behavior of values stored as properties in value objects is the same. For example, suppose vobj1 is a value object with property a:

```
vobj1.a = 8; % Property is set to a value
```

If you copy vobj1 to vobj2, and then change the value of vobj1 property a, you can see that the value of the copied object's property vobj2.a is unaffected:

```
vobj2 =vobj1;
vobj1.a = 5;

vobj2.a
ans =
     8
```

Handle Object Copy Behavior

Suppose you have a handle class called HdClass that defines a property called Data, and that you create an object of this class with the following statement:

```
hobj1 = HdClass(8)
```

Because this statement is not terminated with a semicolon, MATLAB displays information about the object:

```
hobj1 =
  HdClass handle
  Properties:
    Data: 8
```

The variable `hobj1` is a handle that references the object created. Copying `hobj1` to `hobj2` results in another handle (the variable `hobj2`) referring to the same object:

```
hobj2 = hobj1
hobj2 =
  HdClass handle
  Properties:
    Data: 8
```

Because handle objects reference the data contained in their properties, copying an object copies the handle to a new variable name, but the properties still refer to the same data. For example, given that `hobj1` is a handle object with property `Data`:

```
hobj1.Data
ans =
    8
```

When you change the value of `hobj1`'s `Data` property, the value of the copied object's `Data` property also changes:

```
hobj1.Data = 5;

hobj2.Data
ans =
    5
```

Because `hobj2` and `hobj1` are handles to the same object, changing the copy, `hobj2`, also changes the data you access through handle `hobj1`:

```
hobj2.Data = 17;
hobj1.Data
ans =
    17
```

Reassigning Handle Variables

Reassigning a handle variable produces the same result as reassigning any MATLAB variable. When you create a new object and assign it to `hobj1`:

```
hobj1 = HdClass(3.14);
```

`hobj1` references the new object, not the same object referenced previously (and still referenced by `hobj2`).

Clearing Handle Variables

When you `clear` a handle from the workspace, MATLAB removes the variable, but does not removed the object referenced by the handle. Therefore, given `hobj1` and `hobj2`, which both reference the same object, you can clear either handle without affecting the object:

```
hobj1.Data = 2^8;
clear hobj1
hobj2
hobj2 =
  HdClass handle
  Properties:
    Data: 256
```

If you clear both `hobj1` and `hobj2`, then there are no references to the object and MATLAB deletes the object and frees the memory used by that object.

Deleting Handle Objects

To remove an object referenced by any number of handles, you `delete` the object. Given `hobj1` and `hobj2`, which both reference the same object, if you delete either handle, MATLAB deletes the object:

```
hobj1 = HdClass(8);
hobj2 = hobj1;
delete(hobj1)
hobj2
hobj2 =
  deleted HdClass handle
```

See "Destroying Objects" on page 5-32 for more information about object lifecycle.

Modifying Objects

When you pass an object to a function, MATLAB follows pass by value semantics. This means that MATLAB passes a copy of the object to the

function. If you modify the object in the function, MATLAB modifies only the copy of the object. The differences in copy behavior between handle and value classes are important in such cases:

- Value class — The function must return the modified copy of the object to the caller.
- Handle class — The copy refers to the same data as the original object. Therefore, the function does not need to return the modified copy.

See for more information.

More Information About Handle and Value Classes

For information about handle and value classes for class designers, see "Value or Handle Class — Which to Use" in the Object-Oriented Programming documentation.

Testing for Handle or Value Class

If you are writing MATLAB programs that copy objects, you might need to determine if any given object is a handle or value. To determine if an object is a handle object, use the isa function:

```
isa(obj,'handle')
```

For example, the containers.Map class creates a handle object:

```
hobj = containers.Map({'Red Sox','Yankees'}, {'Boston','New York'});
isa(hobj,'handle')
ans =
     1
```

hobj is also a containers.Map object:

```
isa(hobj,'containers.Map')
ans =
     1
```

If you query the class of hobj, you see that it is a containers.Map object:

```
class(hobj)
```

```
ans =
containers.Map
```

The `class` function returns the specific class of an object, whereas `isa` returns `true` for any of the object's superclasses as well. This behavior is consistent with the object-oriented concept that an object is a member of all its superclasses. Therefore, it is true that a `containers.Map` object is a handle object and a `containers.Map` object.

There is no equivalent test for value classes because there is no value base class. If an object is a value object, `isa(object,'handle')` returns `false` (i.e., logical `0`).

See "Map Containers" on page 1-150 for more information on the `containers.Map` class.

Destroying Objects

Object Lifecycle

An object's lifecycle ends when you reassign a new value to that variable, when it is no longer used in a function, or when function execution ends. MATLAB handle classes have a special method called delete that MATLAB calls when a handle object lifecycle ends.

Calling delete on an object explicitly makes all copies of a handle object invalid because it destroys the data associated with the object and frees memory used by deleted objects. MATLAB calls delete automatically so it is not necessary for you to do so. Classes can redefine the handle class delete method to perform other cleanup operations, like closing files or saving data.

Deleting a handle object renders all copies invalid:

```
delete(hobj1)
hobj2.a
??? Invalid or deleted object.
```

Difference Between clear and delete

The handle class delete method removes the handle object, but does not clear the variable name. The clear function removes a variable name, but does not remove the values to which the variable refers. For example, if you have two variables that refer to the same handle object, you can clear either one without affecting the actual object:

```
hobj = containers.Map({'Red Sox','Yankees'}, {'Boston','New York'});
hobj_copy = hobj;
clear hobj
city = hobj_copy('Red Sox')
city =
Boston
```

If you call `clear` on all handle variables that refer to the same handle object, then you have lost access to the object and MATLAB destroys the object. That is, when there are no references to an object, the object ceases to exist.

You can only call `clear` on value objects. MATLAB does not automatically call a value class `delete` method.

6

Error Handling

Error Reporting in a MATLAB Application

In this section...
"Overview" on page 6-2
"Getting an Exception at the Command Line" on page 6-2
"Getting an Exception in Your Program Code" on page 6-3
"Generating a New Exception" on page 6-4

Overview

No matter how carefully you plan and test the programs you write, they may not always run as smoothly as expected when executed under different conditions. It is always a good idea to include error checking in programs to ensure reliable operation under all conditions.

In the MATLAB software, you can decide how your programs respond to different types of errors. You may want to prompt the user for more input, display extended error or warning information, or perhaps repeat a calculation using default values. The error-handling capabilities in MATLAB help your programs check for particular error conditions and execute the appropriate code depending on the situation.

When MATLAB detects a severe fault in the command or program it is running, it collects information about what was happening at the time of the error, displays a message to help the user understand what went wrong, and terminates the command or program. This is called *throwing an exception*. You can get an exception while entering commands at the MATLAB command prompt or while executing your program code.

Getting an Exception at the Command Line

If you get an exception at the MATLAB prompt, you have several options on how to deal with it as described below.

Determine the Fault from the Error Message

Evaluate the error message MATLAB has displayed. Most error messages attempt to explain at least the immediate cause of the program failure. There

is often sufficient information to determine the cause and what you need to do to remedy the situation.

Review the Failing Code

If the function in which the error occurred is implemented as a MATLAB program file, the error message should include a line that looks something like this:

```
surf
```

```
??? Error using ==> surf at 50
Not enough input arguments.
```

The underlined text to the right names the function that threw the error (surf, in this case) and shows the failing line number within that function's program file. Click the underlined text; MATLAB opens the file and positions the cursor at the location in the file where the error originated. You may be able to determine the cause of the error by examining this line and the code that precedes it.

Step Through the Code in the Debugger

You can use the MATLAB Debugger to step through the failing code. Click the underlined error text to open the file in the MATLAB Editor at or near the point of the error. Next, click the hyphen at the beginning of that line to set a breakpoint at that location. When you rerun your program, MATLAB pauses execution at the breakpoint and enables you to step through the program code. The command dbstop on error is also helpful in finding the point of error.

See the documentation on "Editing and Debugging MATLAB Code" for more information.

Getting an Exception in Your Program Code

When you are writing your own program in a program file, you can *catch* exceptions and attempt to handle or resolve them instead of allowing your program to terminate. When you catch an exception, you interrupt the normal termination process and enter a block of code that deals with the faulty situation. This block of code is called a *catch block*.

Some of the things you might want to do in the catch block are:

- Examine information that has been captured about the error.
- Gather further information to report to the user.
- Try to accomplish the task at hand in some other way.
- Clean up any unwanted side effects of the error.

When you reach the end of the catch block, you can either continue executing the program, if possible, or terminate it.

The documentation on "Capturing Information About the Error" on page 6-5 describes how to acquire information about what caused the error, and "Responding to an Exception" on page 6-18 presents some ideas on how to respond to it.

Generating a New Exception

When your program code detects a condition that will either make the program fail or yield unacceptable results, it should throw an exception. This procedure

- Saves information about what went wrong and what code was executing at the time of the error.
- Gathers any other pertinent information about the error.
- Instructs MATLAB to throw the exception.

The documentation on "Capturing Information About the Error" on page 6-5 describes how to use an MException object to capture information about the error, and "Throwing an Exception" on page 6-16 explains how to initiate the exception process.

Capturing Information About the Error

Overview

When the MATLAB software throws an exception, it captures information about what caused the error in a data structure called an MException object. This object is an instance of the MATLAB MException class. You can obtain access to the MException object by *catching* the exception before your program aborts and accessing the object constructed for this particular error via the catch command. When throwing an exception in response to an error in your own code, you will have to create a new MException object and store information about the error in that object.

This section describes the MException class and objects constructed from that class:

Information on how to use this class is presented in later sections on "Responding to an Exception" on page 6-18 and "Throwing an Exception" on page 6-16.

The MException Class

The figure shown below illustrates one possible configuration of an object of the MException class. The object has four properties: identifier, message, stack, and cause. Each of these properties is implemented as a field of the structure that represents the MException object. The stack field is an N-by-1 array of additional structures, each one identifying a function, and line number from the call stack. The cause field is an M-by-1 cell array of MException objects, each representing an exception that is related to the current one.

See "Properties of the MException Class" on page 6-7 for a full description of these properties.

Object Constructor

Any code that detects an error and throws an exception must also construct an MException object in which to record and transfer information about the error. The syntax of the MException constructor is

```
ME = MException(identifier, message)
```

where `identifier` is a MATLAB message identifier of the form

```
component:mnemonic
```

that is enclosed in single quotes, and `message` is a text string, also enclosed in single quotes, that describes the error. The output `ME` is the resulting `MException` object.

If you are responding to an exception rather than throwing one, you do not have to construct an `MException` object. The object has already been constructed and populated by the code that originally detected the error.

Properties of the MException Class

The `MException` class has four properties. Each of these properties is implemented as a field of the structure that represents the `MException` object. Each of these properties is described in the sections below and referenced in the sections on "Responding to an Exception" on page 6-18 and "Throwing an Exception" on page 6-16. All are read-only; their values cannot be changed.

The `MException` properties are:

- `identifier`
- `message`
- `stack`
- `cause`

Repeating the `surf` example shown above, but this time catching the exception, you can see the four properties of the `MException` object structure. (This example uses `try-catch` in an atypical fashion. See the section on "The try-catch Statement" on page 6-18 for more information on using `try-catch`).

```
try
    surf
catch ME
    ME
end
```

Run this at the command line and MATLAB returns the contents of the `MException` object:

```
ME =
 MException object with properties:

    identifier: 'MATLAB:nargchk:notEnoughInputs'
       message: 'Not enough input arguments.'
         stack: [1x1 struct]
         cause: {}
```

The `stack` field shows the filename, function, and line number where the exception was thrown:

```
ME.stack
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

The `cause` field is empty in this case. Each field is described in more detail in the sections that follow.

Message Identifiers

A message identifier is a tag that you attach to an error or warning statement that makes that error or warning uniquely recognizable by MATLAB. You can use message identifiers with error reporting to better identify the source of an error, or with warnings to control any selected subset of the warnings in your programs.

The message identifier is a read-only character string that specifies a *component* and a *mnemonic* label for an error or warning. The format of a simple identifier is

```
component:mnemonic
```

A colon separates the two parts of the identifier: `component` and `mnemonic`. If the identifier uses more than one `component`, then additional colons are required to separate them. A message identifier must always contain at least one colon.

Some examples of message identifiers are

```
MATLAB:rmpath:DirNotFound
MATLAB:odearguments:InconsistentDataType
Simulink:actionNotTaken
TechCorp:OpenFile:notFoundInPath
```

Both the `component` and `mnemonic` fields must adhere to the following syntax rules:

- No white space (space or tab characters) is allowed anywhere in the identifier.
- The first character must be alphabetic, either uppercase or lowercase.
- The remaining characters can be alphanumeric or an underscore.

There is no length limitation to either the `component` or `mnemonic`. The identifier can also be an empty string.

Component Field. The `component` field specifies a broad category under which various errors and warnings can be generated. Common components are a particular product or toolbox name, such as `MATLAB` or `Control`, or perhaps the name of your company, such as `TechCorp` in the preceding example.

You can also use this field to specify a multilevel component. The following statement has a three-level component followed by a mnemonic label:

```
TechCorp:TestEquipDiv:Waveform:obsoleteSyntax
```

The component field enables you to guarantee the uniqueness of each identifier. Thus, while the internal MATLAB code might use a certain warning identifier like `MATLAB:InconsistentDataType`, that does not preclude you from using the same mnemonic, as long as you precede it with a unique component. For example,

```
warning('TechCorp:InconsistentDataType', ...
    'Value %s is inconsistent with existing properties.' ...
    sprocketDiam)
```

Mnemonic Field. The mnemonic field is a string normally used as a tag relating to the particular message. For example, when reporting an error resulting from the use of ambiguous syntax, a simple component and mnemonic such as the following might be appropriate:

```
MATLAB:ambiguousSyntax
```

Message Identifiers in an MException Object. When throwing an exception, create an appropriate identifier and save it to the MException object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

For example,

```
ME = MException('AcctError:NoClient', ...
      'Client name not recognized.');

ME.identifier
ans =
    AcctError:NoClient
```

When responding to an exception, you can extract the message identifier from the MException object as shown here. Using the surf example again,

```
try
    surf
catch ME
    id = ME.identifier
end

id =
    MATLAB:nargchk:notEnoughInputs
```

Text of the Error Message

An error message in MATLAB is a read-only character string issued by the program code and returned in the MException object. This message can assist the user in determining the cause, and possibly the remedy, of the failure.

When throwing an exception, compose an appropriate error message and save it to the MException object at the time you construct the object using the syntax

```
ME = MException(identifier, string)
```

If your message string requires formatting specifications, like those available with the sprintf function, use this syntax for the MException constructor:

```
ME = MException(identifier, formatstring, arg1, arg2, ...)
```

For example,

```
S = 'Accounts';  f1 = 'ClientName';
ME = MException('AcctError:Incomplete', ...
        'Field ''%s.%s'' is not defined.', S, f1);

ME.message
ans =
    Field 'Accounts.ClientName' is not defined.
```

When responding to an exception, you can extract the error message from the MException object as follows:

```
try
    surf
catch ME
    msg = ME.message
end

msg =
    Not enough input arguments.
```

The Call Stack

The stack field of the MException object identifies the line number, function, and filename where the error was detected. If the error occurs in a called function, as in the following example, the stack field contains the line number, function name, and filename not only for the location of the immediate error, but also for each of the calling functions. In this case, stack

is an N-by-1 array, where N represents the depth of the call stack. That is, the stack field displays the function name and line number where the exception occurred, the name and line number of the caller, the caller's caller, etc., until the top-most function is reached.

When throwing an exception, MATLAB stores call stack information in the `stack` field. You cannot write to this field; access is read-only.

For example, suppose you have three functions that reside in two separate files:

```
  mfileA.m
========================
        .
        .
42 function A1(x, y)
43 B1(x, y);




  mfileB.m
========================
        .
        .
 8 function B1(x, y)
 9 B2(x, y)
        .
        .
26 function B2(x, y)
27      .
28      .
29      .
30      .
31 %  Throw exception here
```

Catch the exception in variable ME and then examine the `stack` field:

```
for k=1:length(ME.stack)
    ME.stack(k)
end
```

```
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B2'
    line: 31
ans =
    file: 'C:\matlab\test\mfileB.m'
    name: 'B1'
    line: 9
ans =
    file: 'C:\matlab\test\mfileA.m'
    name: 'A1'
    line: 43
```

The Cause Array

In some situations, it can be important to record information about not only the one command that caused execution to stop, but also other exceptions that your code caught. You can save these additional MException objects in the cause field of the primary exception.

The cause field of an MException is an optional cell array of related MException objects. You must use the following syntax when adding objects to the cause cell array:

```
primaryException = addCause(primaryException, secondaryException)
```

This example attempts to assign an array D to variable X. If the D array does not exist, the code attempts to load it from a MAT-file and then retries assigning it to X. If the load fails, a new MException object (ME3) is constructed to store the cause of the first two errors (ME1 and ME2):

```
try
    X = D(1:25)
catch ME1
    try
        filename = 'test200';
        load(filename);
        X = D(1:25)
    catch ME2
        ME3 = MException('MATLAB:LoadErr', ...
```

```
                        'Unable to load from file %s', filename);
              ME3 = addCause(ME3, ME1);
              ME3 = addCause(ME3, ME2);
         end
    end
```

There are two exceptions in the cause field of ME3:

```
    ME3.cause
    ans =
         [1x1 MException]
         [1x1 MException]
```

Examine the cause field of ME3 to see the related errors:

```
    ME3.cause{:}
    ans =

     MException object with properties:

        identifier: 'MATLAB:UndefinedFunction'
           message: 'Undefined function or method 'D' for input
    arguments of type 'double'.'
             stack: [0x1 struct]
             cause: {}
    ans =

     MException object with properties:

        identifier: 'MATLAB:load:couldNotReadFile'
           message: 'Unable to read file test204: No such file or
    directory.'
             stack: [0x1 struct]
             cause: {}
```

Methods of the MException Class

There are ten methods that you can use with the MException class. The names of these methods are case-sensitive. See the MATLAB function reference pages for more information.

Method Name	Description
addCause	Append an MException to the cause field of another MException.
disp	Display an MException object.
eq	Compare MException objects for equality.
getReport	Return a formatted message based on the current exception.
isequal	Compare MException objects for equality.
last	Return the last uncaught exception. This is a static method.
ne	Compare MException objects for inequality.
rethrow	Reissue an exception that has previously been caught.
throw	Issue an exception.
throwAsCaller	Issue an exception, but omit the current stack frame from the stack field.

Throwing an Exception

When your program detects a fault that will keep it from completing as
expected or will generate erroneous results, you should halt further execution
and report the error by throwing an exception. The basic steps to take are

1 Detect the error. This is often done with some type of conditional statement,
such as an if statement that checks the output of the current operation.

2 Construct an MException object to represent the error. Add a message
identifier string and error message string to the object when calling the
constructor.

3 If there are other exceptions that may have contributed to the current error,
you can store the MException object for each in the cause field of a single
MException that you intend to throw. Use the addCause method for this.

4 Use the throw or throwAsCaller function to have the MATLAB software
issue the exception. At this point, MATLAB stores call stack information in
the stack field of the MException, exits the currently running function,
and returns control to either the keyboard or an enclosing catch block in a
calling function.

This example illustrates throwing an exception using the steps just described:

```
function check_results(resultsArr, dataFile)
minValue = 0.09;   maxValue = 2.14;

% 1) Detect the error.
if any(resultsArr < minValue) || any(resultsArr > maxValue)

    % 2) Construct an MException object to represent the error.
    err = MException('ResultChk:OutOfRange', ...
        'Resulting value is outside expected range');
    fileInfo = dir(dataFile);

    % 3) Store any information contributing to the error.
    if datenum(fileInfo.date) < datenum('OctO9','mmmyy')
        errCause = MException('ResultChk:BadInput', ...
            'Input file %s is out of date.', dataFile);
```

```
        err = addCause(err, errCause);
    end

    % 4) Throw the exception to stop execution and display
          an error message.
    throw(err)
end
```

If the program detects the `OutOfRange` condition, the `throw(err)` statement throws an exception at the end. If the `BadInput` condition is also detected, the program also displays this as the cause:

```
resultsArr = [1.63, 2.05, 0.91, 2.16, 1.5, 2.11 0.72];

check_results(resultsArr, 'run33.dat')
??? Error using ==> check_results at 12
Resulting value is outside expected range

Caused by:
    Input file run33.dat is out of date.
```

Responding to an Exception

In this section...
"Overview" on page 6-18
"The try-catch Statement" on page 6-18
"Suggestions on How to Handle an Exception" on page 6-20

Overview

As stated earlier, the MATLAB software, by default, terminates the currently running program when an exception is thrown. If you catch the exception in your program, however, you can capture information about what went wrong, and deal with the situation in a way that is appropriate for the particular condition. This requires a `try-catch` statement.

This section covers the following topics:

The try-catch Statement

When you have statements in your code that could generate undesirable results, put those statements into a `try-catch` block that catches any errors and handles them appropriately.

A `try-catch` statement looks something like the following pseudocode. It consists of two parts:

- A `try` block that includes all lines between the `try` and `catch` statements.

- A `catch` block that includes all lines of code between the `catch` and `end` statements.

```
      try
         Perform one ...
            or more operations
   A    catch ME
         Examine error info in exception object ME
         Attempt to figure out what went wrong
         Either attempt to recover, or clean up and abort
```

```
          end

    B   Program continues
```

The program executes the statements in the `try` block. If it encounters an error, it skips any remaining statements in the `try` block and jumps to the start of the `catch` block (shown here as point A). If all operations in the `try` block succeed, then execution skips the `catch` block entirely and goes to the first line following the `end` statement (point B).

Specifying the `try`, `catch`, and `end` commands and also the code of the `try` and `catch` blocks on separate lines is recommended. If you combine any of these components on the same line, separate them with commas:

```
try, surf, catch ME, ME.stack, end
ans =
    file: 'matlabroot\toolbox\matlab\graph3d\surf.m'
    name: 'surf'
    line: 54
```

Note You cannot define nested functions within a `try` or `catch` block.

The Try Block

On execution, your code enters the `try` block and executes each statement as if it were part of the regular program. If no errors are encountered, MATLAB skips the `catch` block entirely and continues execution following the `end` statement. If any of the `try` statements fail, MATLAB immediately exits the `try` block, leaving any remaining statements in that block unexecuted, and enters the `catch` block.

The Catch Block

The `catch` command marks the start of a `catch` block and provides access to a data structure that contains information about what caused the exception. This is shown as the variable `ME` in the preceding pseudocode. This data structure is an object of the MATLAB `MException` class. When an exception occurs, MATLAB constructs an instance of this class and returns it in the `catch` statement that handles that error.

You are not required to specify any argument with the `catch` statement. If you do not need any of the information or methods provided by the `MException` object, just specify the `catch` keyword alone.

The `MException` object is constructed by internal code in the program that fails. The object has properties that contain information about the error that can be useful in determining what happened and how to proceed. The `MException` object also provides access to methods that enable you to respond to the exception. See the section on"The MException Class" on page 6-5 to find out more about the `MException` class.

Having entered the `catch` block, MATLAB executes the statements in sequence. These statements can attempt to

- Attempt to resolve the error.
- Capture more information about the error.
- Switch on information found in the `MException` object and respond appropriately.
- Clean up the environment that was left by the failing code.

The `catch` block often ends with a `rethrow` command. The `rethrow` causes MATLAB to exit the current function, keeping the call stack information as it was when the exception was first thrown. If this function is at the highest level, that is, it was not called by another function, the program terminates. If the failing function was called by another function, it returns to that function. Program execution continues to return to higher level functions, unless any of these calls were made within a higher-level `try` block, in which case the program executes the respective catch block.

More information about the `MException` class is provided in the section "Capturing Information About the Error" on page 6-5.

Suggestions on How to Handle an Exception

The following example reads the contents of an image file. The `try` block attempts to open and read the file. If either the open or read fails, the program catches the resulting exception and saves the `MException` object in the variable ME1.

The catch block in the example checks to see if the specified file could not be found. If so, the program allows for the possibility that a common variation of the filename extension (e.g., jpeg instead of jpg) was used by retrying the operation with a modified extension. This is done using a try-catch statement nested within the original try-catch.

```
function d_in = read_image(filename)
[path name ext] = fileparts(filename);
try
   fid = fopen(filename, 'r');
   d_in = fread(fid);
catch ME1
   % Get last segment of the error message identifier.
   idSegLast = regexp(ME1.identifier, '(?<=:)\w+$', 'match');

   % Did the read fail because the file could not be found?
   if strcmp(idSegLast, 'InvalidFid') && ...
      ~exist(filename, 'file')

      % Yes. Try modifying the filename extension.
      switch ext
      case '.jpg'     % Change jpg to jpeg
          filename = strrep(filename, '.jpg', '.jpeg')
      case '.jpeg'    % Change jpeg to jpg
          filename = strrep(filename, '.jpeg', '.jpg')
      case '.tif'     % Change tif to tiff
          filename = strrep(filename, '.tif', '.tiff')
      case '.tiff'    % Change tiff to tif
          filename = strrep(filename, '.tiff', '.tif')
      otherwise
          fprintf('File %s not found\n', filename);
          rethrow(ME1);
      end

      % Try again, with modifed filenames.
      try
         fid = fopen(filename, 'r');
         d_in = fread(fid);
      catch ME2
         fprintf('Unable to access file %s\n', filename);
```

```
                ME2 = addCause(ME2, ME1);
                rethrow(ME2)
            end
        end
    end
```

This example illustrates some of the actions that you can take in response to an exception:

- Compare the `identifier` field of the `MException` object against possible causes of the error.

- Use a nested `try-catch` statement to retry the open and read operations using a known variation of the filename extension.

- Display an appropriate message in the case that the file truly does not exist and then rethrow the exception.

- Add the first `MException` object to the `cause` field of the second.

- Rethrow the exception. This stops program execution and displays the error message.

Cleaning up any unwanted results of the error is also advisable. For example, your program may have allocated a significant amount of memory that it no longer needs.

Warnings

Reporting a Warning

Like error, the warning function alerts the user of unexpected conditions detected when running a program. However, warning does not halt the execution of the program. It displays the specified warning message and then continues.

Use warning in your code to generate a warning message during execution. Specify the message string as the input argument to warning. For example,

```
warning('Input must be a string')
```

Warnings also differ from errors in that you can disable any warnings that you do not want to see. You do this by invoking warning with certain control parameters. See "Warning Control" on page 6-25 for more information.

Formatted Message Strings

The warning message string you specify can contain formatting conversion characters, such as those used with the MATLAB sprintf function. Make the warning string the first argument, and add any variables used by the conversion as subsequent arguments.

```
warning('formatted_warningmsg', arg1, arg2, ...)
```

For example, if your program cannot process a given parameter, you might report a warning with

```
warning('Ambiguous parameter name, "%s".', param)
```

MATLAB converts special characters like %d and %s in the warning message string only when you specify more than one input argument with warning. See "Formatted Message Strings" on page 6-23 for information.

Message Identifiers

Use a message identifier argument with `warning` to attach a unique tag to a warning message. MATLAB uses this tag to better identify the source of a warning. The first argument in this example is the message identifier.

```
warning('MATLAB:paramAmbiguous', ...
        'Ambiguous parameter name, "%s".', param)
```

See "Warning Control Statements" on page 6-27 for more information on how to use identifiers with warnings.

Identifying the Cause

The `lastwarn` function returns a string containing the last warning message issued by MATLAB. Use this to enable your program to identify the cause of a warning that has just been issued. To return the most recent warning message to the variable `warnmsg`, type

```
warnmsg = lastwarn;
```

You can also change the text of the last warning message with a new message or with an empty string as shown here:

```
lastwarn('newwarnmsg');   % Replace last warning with new string
lastwarn('');             % Replace last warning with empty string
```

Warning Control

In this section...
"Overview" on page 6-25
"Warning Statements" on page 6-26
"Warning Control Statements" on page 6-27
"Output from Control Statements" on page 6-30
"Saving and Restoring State" on page 6-32
"Backtrace and Verbose Modes" on page 6-33

Overview

The MATLAB software gives you the ability to control what happens when a warning is encountered during program execution. Options that are available include

- Display selected warnings.
- Ignore selected warnings.
- Stop in the debugger when a warning is invoked.
- Display the stack trace after a warning is invoked.

Depending on how you set your warning controls, you can have these actions affect all warnings in your code, specific warnings that you select, or just the most recently invoked warning.

Setting up this system of warning control involves several steps.

1 Start by determining the scope of the control you need for the warnings generated by your code. Do you want the control operations to affect all the warnings in your code at once, or do you want to be able to control certain warnings separately?

2 If the latter is true, you will need to identify those warnings you want to selectively control. This requires going through your code and attaching unique *message identifiers* to each of those warnings. If, on the other

hand, you do not require that fine a granularity of control, the warning statements in your code need no message identifiers.

3 When you are ready to run your programs, use the MATLAB warning control statements to exercise the desired controls on all or selected warnings. Include message identifiers in these control statements when selecting specific warnings to act upon.

Warning Statements

The warning statements you put into your code must contain the string to be displayed when the warning is incurred, and may also contain a message identifier. If you are not planning to use warning control or if you do not need to single out certain warnings for control, you need to specify only the message string. Use the syntax shown in "Warnings" on page 6-23. Valid formats are

```
warning('warnmsg')
warning('formatted_warnmsg', arg1, arg2, ...)
```

Attaching an Identifier to the Warning Statement

If you want to be able to apply control statements to specific warnings, you need to include a message identifier in the warning statements you wish to control. The message identifier must be the first argument in the statement. Valid formats are

```
warning('msg_id', 'warnmsg')
warning('msg_id', 'formatted_warnmsg', arg1, arg2, ...)
```

See "Message Identifiers" on page 6-8 for information on how to specify the msg_id argument.

Note When you specify more than one input argument with warning, MATLAB treats the warnmsg string as if it were a formatted_warnmsg. This is explained in "Formatted Message Strings" on page 6-23.

Warning Control Statements

Once you have the warning statements in your program file and are ready to execute it, you tell MATLAB how to act on these warnings by issuing control statements. These statements place the specified warning(s) into a desired state and have the format

```
warning state msg_id
```

Control statements can return information on the state of selected warnings if you assign the output to a variable, as shown below. See "Output from Control Statements" on page 6-30.

```
s = warning('state', 'msg_id');
```

Warning States

There are three possible values for the `state` argument of a warning control statement.

State	Description
on	Enable the display of selected warning message.
off	Disable the display of selected warning message.
query	Display the current state of selected warning.

Message Identifiers

In addition to the message identifiers already discussed, there are three other identifiers that you can use in control statements only.

Identifier	Description
msg_id string	Set selected warning to the specified state.
all	Set all warnings to the specified state.
last	Set only the last displayed warning to the specified state.

> **Note** MATLAB starts up with all warnings enabled, except for those
> displayed in response to the command, `warning('query', 'all')`.

When warnings are disabled, the dbstop **if warning** commands have no
effect. If warnings are disabled for specific message identifiers, the dbstop **if
warning** identifier has no effect for those identifiers.

Retrieving a Message Identifier from a Warning. If you get a warning
and you would like to know what the message identifier is for that warning,
you can retrieve the identifier from the second output of the lastwarn
function. The following example generates a warning when it attempts to
concatenate two unlike integer types together:

```
warning on all;

A = [int8(150), int16(300)];
Warning: Concatenation with dominant (left-most) integer class
   may overflow other operands on conversion to return class.
```

If you are already aware of the consequences of this command and do not
want to see this warning message displayed every time you run your program,
you can disable the warning message. To identify the warning to disable, use
the following commands to acquire the message identifier:

```
warnStruct = warning('query', 'last');
msgid_integerCat = warnStruct.identifier
msgid_integerCat =
   MATLAB:concatenation:integerInteraction
```

Once you have the identifier, you can use it to disable this one particular
message:

```
warning('off', msgid_integerCat);
```

Try the command again

```
A = [int8(150), int16(300)]
A =
  127  127
```

Turn the message back on again, if you need to, as shown here:

```
warning('on', msgid_intcatwarn);
MATLAB:nonScalarConditional
```

Enabling and Disabling Selected Warnings. Enable just the
actionNotTaken warning from Simulink by first turning off all warnings and
then setting just that warning to on.

```
warning off all
warning on Simulink:actionNotTaken
```

Next, use query to determine the current state of all warnings. It
reports that you have set all warnings to off, with the exception of
Simulink:actionNotTaken.

```
warning query all
The default warning state is 'off'. Warnings not set to the
default are

   State  Warning Identifier

      on  Simulink:actionNotTaken
```

Enabling and Disabling All Warnings. You can enable or disable all
warnings using the 'all' identifier. Using the query option shows the result:

```
warning on all

warning query all
The default warning state is 'on'. Warnings not set to the
default are

   State  Warning Identifier

      off  MATLAB:nonScalarConditional
```

Note that, in this case, there is one warning that does not take on the default
state when that state is changed from off to on. This is intentional. If so
desired, you can force this or any individual warning to either the on or off
state by specifying the message identifier in the command:

```
warning on MATLAB:nonScalarConditional

warning query all
   All warnings have the state 'on'.
```

Disabling the Most Recent Warning. Evaluating inv on zero displays a warning message. Turn off the most recently invoked warning with `warning off last`.

```
inv(0)
Warning: Matrix is singular to working precision.
ans =
   Inf

warning off last

inv(0)                % No warning is displayed this time
ans =
   Inf
```

Output from Control Statements

The warning function, when used in a control statement, returns a MATLAB structure array containing the previous state of the selected warning(s). Use the following syntax to return this information in structure array `s`:

```
s = warning('state', 'msg_id');
```

You must type the command using the MATLAB function format; parentheses and quotation marks are required.

Note MATLAB does not display warning output if you do not assign the output to a variable.

The next example turns off InconsistentDataType warnings for the MATLAB:odearguments component, and returns the identifier and previous state in a 1-by-1 structure array.

```
MATLAB:odearguments:InconsistentDataType
```

```
s = warning('off','MATLAB:odearguments:InconsistentDataType')
s =
    identifier: 'MATLAB:odearguments:InconsistentDataType'
         state: 'on'
```

You can use output variables with any type of warning control statement. If you just want to collect the information but do not want to change state, simply perform a query on the warning(s). MATLAB returns the current state of those warnings selected by the message identifier.

```
s = warning('query', 'msg_id');
```

If you want to change state, but save the former state so you can restore it later, use the return structure array to save that state. The following example does an implicit query, returning state information in s, and then turns on all warnings.

```
s = warning('on', 'all');
```

See "Saving and Restoring State" on page 6-32, for more information on restoring the former state of warnings.

Output Structure Array

Each element of the structure array returned by warning contains two fields.

Field Name	Description
identifier	Message identifier string, 'all', or 'last'
state	State of warning(s) prior to invoking this control statement

If you query for the state of just one warning, using a message identifier or 'last' in the command, MATLAB returns a one-element structure array. The identifier field contains the selected message identifier, and the state field holds the current state of that warning:

```
s = warning('query','last')
s =
    identifier: 'MATLAB:odearguments:InconsistentDataType'
         state: 'on'
```

If you query for the state of all warnings, using `'all'` in the command, MATLAB returns a structure array having one or more elements:

- The first element of the array always represents the default state. (This is the state set by the last `warning on|off all` command.)

- Each other element of the array represents a warning that is in a state different from the default.

```
warning off all
warning on MATLAB:odearguments:InconsistentDataType
warning on MATLAB:rmpath:DirNotFound

s = warning('query', 'all')
s =
   3x1 struct array with fields:
       identifier
       state

s(1)
ans =
   identifier: 'all'
        state: 'off'

s(2)
ans =
    identifier: 'MATLAB:odearguments:InconsistentDataType'
         state: 'on'
s(3)
ans =
    identifier: 'MATLAB:rmpath:DirNotFound'
         state: 'on'
```

Saving and Restoring State

To temporarily change the state of some warnings and then later return to your original settings, save the original state in a structure array and then restore it from that array. You can save and restore the state of all of your warnings or just one that you select with a message identifier.

To save the current warning state, assign the output of a warning control statement, as discussed in "Output from Control Statements" on page 6-30. The following statement saves the current state of all warnings in structure array s:

```
s = warning('query', 'all');
```

To restore state from s, use the syntax shown below. Note that the MATLAB function format (enclosing arguments in parentheses) is required.

```
warning(s)
```

Example 1 — Performing an Explicit Query

Perform a query of all warnings to save the current state in structure array s:

```
s = warning('query', 'all');
```

Then, after doing some work that includes making changes to the state of some warnings, restore the original state of all warnings:

```
warning(s)
```

Example 2 — Performing an Implicit Query

Turn on one particular warning, saving the previous state of this warning in s. Remember that this nonquery syntax (where state equals on or off) performs an implicit query prior to setting the new state:

```
s = warning('on', 'Control:parameterNotSymmetric');
```

Restore the state of that one warning when you are ready, with

```
warning(s)
```

Backtrace and Verbose Modes

In addition to warning messages, there are two *modes* that can be enabled or disabled with a warning control statement. These modes are shown here.

Mode	Description	Default
verbose	Display a message on how to suppress the warning.	off (terse)
backtrace	Display an stack trace after a warning is invoked.	on (enabled)

The syntax for this type of control statement is as follows, where state, in this case, can be only on, off, or query:

```
warning state mode
```

Note that there is no need to include a message identifier with this type of control statement. All enabled warnings are affected by the this type of control statement.

Note You cannot save and restore the current state of the backtrace or verbose modes as you can with other states.

Example 1 — Enabling Verbose Warnings

When you enable verbose warnings, MATLAB displays an extra line of information with each warning that tells you how to suppress it:

Turn on all warnings, disable backtrace (if you have just run the previous example), and enable verbose warnings:

```
warning on all
warning off backtrace
warning on verbose
```

Create a function that tests a condition and displays a warning message based on the input:

```
function testArrayMax(arr, max)
exceedMax = find(arr > max);
if any(exceedMax)
    warning('TestEnv:InvalidInput', ...
        'Values in array "%s" exceed the maximum.', ...
```

```
            inputname(1))
end
```

Call the function to find out how to suppress warnings that might be generated by that function. Note the last line displayed here:

```
A = [1287, 5010, 2759];

testArrayMax(A, 5000)
Warning: Values in array "A" exceed the maximum.
(Type "warning off TestEnv:InvalidInput" to suppress this warning.)
```

Use the message identifier `TestEnv:InvalidInput` to disable only this warning, and run the function again. This time the warning message is not displayed:

```
warning off TestEnv:InvalidInput
testArrayMax(A, 5000)
```

Example 2 — Displaying a Stack Trace on a Specific Warning

It can be difficult to locate the source of a warning when it is generated from code buried in several levels of function calls. This example generates a warning within a function that is nested several levels deep within the primary function in file `isValidArray.m`:

```
function isValidArray(A)
max = 5000;
nestFun_1
    function nestFun_1
        nestFun_2
        function nestFun_2
            testArrayMax(A, max);
        end
    end
end
```

After enabling all warnings, run the program. Due to the value of `A(2)`, the function generates a warning:

```
warning on all
warning off verbose

A = [1287, 5010, 2759];

isValidArray(A)
Warning: Values in array "A" exceed the maximum.
```

In a function of this size, it is not difficult to find the cause of the warning, but in a file of several hundred lines, this could take some time. To simplify the debug process, enable backtrace mode. In this mode, MATLAB reports which function generated the warning (testArrayMax), the line number of the attempted operation (line 4), the sequence of function calls that led up to the execution of the function (from isValidArray to nestFun_1 to nestFun_2 and finally to testArrayMax), and the line at which each of these function calls were made (3, 5, 7, and 4):

```
warning on backtrace

callArrayMax(A)
Warning: Values in array "A" exceed the maximum.
> In testArrayMax at 4
  In isValidArray>nestFun_1/nestFun_2 at 7
  In isValidArray>nestFun_1 at 5
  In isValidArray at 3
```

Debugging Errors and Warnings

You can direct the MATLAB software to temporarily stop the execution of an program in the event of a run-time error or warning, at the same time opening a debug window paused at the line that generated the error or warning. This enables you to examine values internal to the program and determine the cause of the error.

Use the `dbstop` function to have MATLAB stop execution and enter debug mode when any function you subsequently run produces a run-time error or warning. There are three types of such breakpoints that you can set.

Command	Description
`dbstop if all error`	Stop on any error.
`dbstop if error`	Stop on any error not detected within a `try-catch` block.
`dbstop if warning`	Stop on any warning.

In all three cases, the file you are trying to debug must be in a folder that is on the search path or in the current folder.

You cannot resume execution after an error; use `dbquit` to exit from the Debugger. To resume execution after a warning, use `dbcont` or `dbstep`.

7

Program Scheduling

Using a MATLAB Timer Object

In this section...
"Overview" on page 7-2
"Example: Displaying a Message" on page 7-3

Overview

The MATLAB software includes a timer object that you can use to schedule the execution of MATLAB commands. This section describes how you can create timer objects, start a timer running, and specify the processing that you want performed when a timer fires. A timer is said to *fire* when the amount of time specified by the timer object elapses and the timer object executes the commands you specify.

To use a timer, perform these steps:

1 Create a timer object.

 You use the `timer` function to create a timer object. See "Creating Timer Objects" on page 7-5 for more information.

2 Specify which MATLAB commands you want executed when the timer fires and control other aspects of timer object behavior.

 You use timer object properties to specify this information. To learn about all the properties supported by the timer object, see "Working with Timer Object Properties" on page 7-7. (You can also set timer object properties when you create them, in step 1.)

3 Start the timer object.

 After you create the timer object, you must start it, using either the `start` or `startat` function. See "Starting and Stopping Timers" on page 7-10 for more information.

4 Delete the timer object when you are done with it.

After you are finished using a timer object, you should delete it from memory. See "Deleting Timer Objects from Memory" on page 7-23 for more information.

Note The specified execution time and the actual execution of a timer can vary because timer objects work in the MATLAB single-threaded execution environment. The length of this time lag is dependent on what other processing MATLAB is performing. To force the execution of the callback functions in the event queue, include a call to the drawnow function in your code. The drawnow function flushes the event queue.

Example: Displaying a Message

The following example sets up a timer object that executes a MATLAB command string after 10 seconds elapse. The example creates a timer object, specifying the values of two timer object properties, TimerFcn and StartDelay. TimerFcn specifies the timer callback function. This is the MATLAB command string or program file that you want to execute when the timer fires. In the example, the timer callback function sets the value of the MATLAB workspace variable stat and executes the MATLAB disp command. The StartDelay property specifies how much time elapses before the timer fires.

After creating the timer object, the example uses the start function to start the timer object. (The additional commands in this example are included to illustrate the timer but are not required for timer operation.)

```
t = timer('TimerFcn', 'stat=false; disp(''Timer!'')',...
                'StartDelay',10);
start(t)

stat=true;
while(stat==true)
  disp('.')
  pause(1)
end
```

When you execute this code, it produces this output:

```
.
.
.
.
.
.
.
.
Timer!

delete(t) % Always delete timer objects after using them.
```

Creating Timer Objects

In this section...
"Creating the Object" on page 7-5
"Naming the Object" on page 7-6

Creating the Object

To use a timer in MATLAB, you must create a timer object. The timer object represents the timer in MATLAB, supporting various properties and functions that control its behavior.

To create a timer object, use the `timer` function. This creates a valid timer object with default values for most properties. The following shows an example of the default timer object and its summary display:

```
t = timer
Timer Object: timer-1

    Timer Settings
       ExecutionMode: singleShot
              Period: 1
             BusyMode: drop
              Running: off

    Callbacks
             TimerFcn: ''
             ErrorFcn: ''
             StartFcn: ''
              StopFcn: ''
```

MATLAB names the timer object `timer-1`. (See "Naming the Object" on page 7-6 for more information.)

To specify the value of timer object properties after you create it, you can use the `set` function. This example sets the value of the `TimerFcn` property and the `StartDelay` property. For more information about timer object properties, see "Working with Timer Object Properties" on page 7-7.

```
set(t,'TimerFcn',@(x,y)disp('Hello World!'),'StartDelay',5)
```

You can also set timer object properties when you create the timer object by specifying property name and value pairs as arguments to the `timer` function. The following example sets the same properties at object creation time:

```
t = timer('TimerFcn', @(x,y)disp('Hello World!'),'StartDelay',5);
```

Always delete timer objects when you are done using them. See "Deleting Timer Objects from Memory" on page 7-23 for more information.

Naming the Object

MATLAB assigns a name to each timer object you create. This name has the form `'timer-i'`, where *i* is a number representing the total number of timer objects created this session.

For example, the first time you call the timer function to create a timer object, MATLAB names the object `timer-1`. If you call the timer function again to create another timer object, MATLAB names the object `timer-2`.

MATLAB keeps incrementing the number associated with each timer object it creates, even if you delete the timer objects you already created. For example, if you delete the first two timer objects and create a new object, MATLAB names it `timer-3`, even though the other two timer objects no longer exist in memory. To reset the numeric part of timer object names to 1, execute the `clear classes` command.

Working with Timer Object Properties

To get information about timer object properties, see the timer function reference page.

Retrieving the Value of Timer Object Properties

The timer object supports many properties that provide information about the current state of the timer object and control aspects of its functioning. To retrieve the value of a timer object property, you can use the get function or use subscripts (dot notation) to access the field.

The following example uses the set function to retrieve the value of the ExecutionMode property:

```
t = timer;

tmode = get(t,'ExecutionMode')

tmode =

singleShot
```

The following example uses dot notation to retrieve the value of the ExecutionMode property:

```
tmode = t.ExecutionMode

tmode =

singleShot
```

To view a list of all the properties of a timer object, use the get function, specifying the timer object as the only argument:

```
get(t)
        AveragePeriod: NaN
              BusyMode: 'drop'
               ErrorFcn: ''
         ExecutionMode: 'singleShot'
         InstantPeriod: NaN
                   Name: 'timer-4'
      ObjectVisibility: 'on'
                 Period: 1
                Running: 'off'
            StartDelay: 0
               StartFcn: ''
                StopFcn: ''
                     Tag: ''
         TasksExecuted: 0
         TasksToExecute: Inf
                TimerFcn: ''
                    Type: 'timer'
                UserData: []
```

Setting the Value of Timer Object Properties

To set the value of a timer object property, use the set function or subscripted assignment (dot notation). You can also set timer object properties when you create the timer object. For more information, see "Creating Timer Objects" on page 7-5.

The following example uses both methods to assign values to timer object properties. The example creates a timer that, once started, displays a message every second until you stop it with the stop command.

1 Create a timer object.

```
t = timer;
```

2 Assign values to timer object properties using the set function.

```
set(t,'ExecutionMode','fixedRate','BusyMode','drop','Period',1);
```

3 Assign a value to the timer object TimerFcn property using dot notation.

```
t.TimerFcn = @(x,y)disp('Processing...');
```

4 Start the timer object. It displays a message at 1-second intervals.

```
start(t)
```

5 Stop the timer object.

```
stop(t)
```

6 Delete timer objects after you are done using them.

```
delete(t)
```

Viewing a List of All Settable Properties

To view a list of all timer object properties that can have values assigned to them (in contrast to the read-only properties), use the set function, specifying the timer object as the only argument.

The display includes the values you can use to set the property if, like the BusyMode property, the property accepts an enumerated list of values.

```
t = timer;

set(t)
    BusyMode: [ {drop} | queue | error ]
    ErrorFcn: string -or- function handle -or- cell array
    ExecutionMode: [{singleShot} | fixedSpacing | fixedDelay | fixedRate]
    Name
    ObjectVisibility: [ {on} | off ]
    Period
    StartDelay
    StartFcn: string -or- function handle -or- cell array
    StopFcn: string -or- function handle -or- cell array
    Tag
    TasksToExecute
    TimerFcn: string -or- function handle -or- cell array
    UserData
```

Starting and Stopping Timers

In this section...

Note Because the timer works within the MATLAB single-threaded environment, it cannot guarantee execution times or execution rates.

Starting a Timer

To start a timer object, call the start function, specifying the timer object as the only argument. The start function starts a timer object running; the amount of time the timer runs is specified in seconds in the StartDelay property.

This example creates a timer object that displays a greeting after 5 seconds elapse.

1 Create a timer object, specifying values for timer object properties.

```
t = timer('TimerFcn',@(x,y)disp('Hello World!'),'StartDelay', 5);
```

2 Start the timer object.

```
start(t)
```

3 Delete the timer object after you are finished using it.

```
delete(t);
```

Starting a Timer at a Specified Time

To start a timer object and specify a date and time for the timer to fire, (rather than specifying the number of seconds to elapse), use the startat function. This function starts a timer object and allows you to specify the date, hour,

minute, and second when you want to the timer to execute. You specify the time as a MATLAB serial date number or as a specially formatted date text string.

This example creates a timer object that displays a message after an hour has elapsed. The `startat` function starts the timer object running and calculates the value of the `StartDelay` property based on the time you specify.

```
t2=timer('TimerFcn',@(x,y)disp('It has been an hour now'));
startat(t2,now+1/24);
```

Stopping Timer Objects

Once started, the timer object stops running if one of the following conditions apply:

- The timer function callback (`TimerFcn`) has been executed the number of times specified in the `TasksToExecute` property.

- An error occurred while executing a timer function callback (`TimerFcn`).

You can also stop a timer object by using the `stop` function, specifying the timer object as the only argument. The following example illustrates stopping a timer object:

1 Create a timer object.

```
t = timer('TimerFcn',@(x,y)disp('Hello World!'), ...
          'StartDelay', 100);
```

2 Start it running.

```
start(t)
```

3 Check the state of the timer object after starting it.

```
get(t,'Running')

ans =

on
```

4 Stop the timer using the `stop` command and check the state again. When a timer stops, the value of the `Running` property of the timer object is set to `'off'`.

```
stop(t)

get(t,'Running')

ans =

off
```

5 Delete the timer object when you are finished using it.

```
delete(t)
```

Note The timer object can execute a callback function that you specify when it starts or stops. See "Creating and Executing Callback Functions" on page 7-14.

Blocking the MATLAB Command Line

By default, when you use the `start` or `startat` function to start a timer object, the function returns control to the command line immediately. For some applications, you might prefer to block the command line until the timer fires. To do this, call the `wait` function right after calling the `start` or `startat` function.

1 Create a timer object.

```
t = timer('StartDelay', 5,'TimerFcn', ...
          @(x,y)disp('Hello World!'));
```

2 Start the timer object running.

```
start(t)
```

3 After the `start` function returns, call the `wait` function immediately. The `wait` function blocks the command line until the timer object fires.

```
wait(t)
```

4 Delete the timer object after you are finished using it.

```
delete(t)
```

Creating and Executing Callback Functions

In this section...
"Associating Commands with Timer Object Events" on page 7-14
"Creating Callback Functions" on page 7-15
"Specifying the Value of Callback Function Properties" on page 7-17

Note Callback function execution might be delayed if the callback involves a CPU-intensive task such as updating a figure.

Associating Commands with Timer Object Events

The timer object supports properties that let you specify the MATLAB commands that execute when a timer fires, and for other timer object events, such as starting, stopping, or when an error occurs. These are called *callbacks*. To associate MATLAB commands with a timer object event, set the value of the associated timer object callback property.

The following diagram shows when the events occur during execution of a timer object and give the names of the timer object properties associated with each event. For example, to associate MATLAB commands with a start event, assign a value to the StartFcn callback property. Error callbacks can occur at any time.

Timer Object Events and Related Callback Function

Creating Callback Functions

When the time period specified by a timer object elapses, the timer object executes one or more MATLAB functions of your choosing. You can specify the functions directly as the value of the callback property. You can also put the commands in a function file and specify the function as the value of the callback property.

Specifying Callback Functions Directly

This example creates a timer object that displays a greeting after 5 seconds. The example specifies the value of the TimerFcn callback property directly, putting the commands in a text string.

```
t = timer('TimerFcn',@(x,y)disp('Hello World!'),'StartDelay',5);
```

Note When you specify the callback commands directly as the value of the callback function property, the commands are evaluated in the MATLAB workspace.

Putting Commands in a Callback Function

Instead of specifying MATLAB commands directly as the value of a callback property, you can put the commands in a MATLAB program file and specify the file as the value of the callback property.

When you create a callback function, the first two arguments must be a handle to the timer object and an event structure. An event structure contains two fields: Type and Data. The Type field contains a text string that identifies the type of event that caused the callback. The value of this field can be any of the following strings: 'StartFcn', 'StopFcn', 'TimerFcn', or 'ErrorFcn'. The Data field contains the time the event occurred.

In addition to these two required input arguments, your callback function can accept application-specific arguments. To receive these input arguments, you must use a cell array when specifying the name of the function as the value of a callback property. For more information, see "Specifying the Value of Callback Function Properties" on page 7-17.

Example: Writing a Callback Function

This example implements a simple callback function that displays the type of event that triggered the callback and the time the callback occurred. To illustrate passing application-specific arguments, the example callback function accepts as an additional argument a text string and includes this text string in the display output. To see this function used with a callback property, see "Specifying the Value of Callback Function Properties" on page 7-17.

```
function my_callback_fcn(obj, event, string_arg)

txt1 = ' event occurred at ';
txt2 = string_arg;

event_type = event.Type;
```

```
event_time = datestr(event.Data.time);

msg = [event_type txt1 event_time];
disp(msg)
disp(txt2)
```

Specifying the Value of Callback Function Properties

You associate a callback function with a specific event by setting the value of the appropriate callback property. You can specify the callback function as a cell array or function handle. If your callback function accepts additional arguments, you must use a cell array.

The following table shows the syntax for several sample callback functions and describes how you call them.

Callback Function Syntax	How to Specify as a Property Value for Object t
`function myfile`	`t.StartFcn = @myfile`
`function myfile(obj, event)`	`t.StartFcn = @myfile`
`function myfile(obj, event, arg1, arg2)`	`t.StartFcn = {@myfile, 5, 6}`

This example illustrates several ways you can specify the value of timer object callback function properties, some with arguments and some without. To see the code of the callback function, my_callback_fcn, see "Example: Writing a Callback Function" on page 7-16:

1 Create a timer object.

```
t = timer('StartDelay', 4, 'Period', 4, 'TasksToExecute', 2, ...
          'ExecutionMode', 'fixedRate');
```

2 Specify the value of the StartFcn callback. Note that the example specifies the value in a cell array because the callback function needs to access arguments passed to it:

```
t.StartFcn = {@my_callback_fcn, 'My start message'};
```

3 Specify the value of the `StopFcn` callback. Again, the value is specified in a cell array because the callback function needs to access the arguments passed to it:

```
t.StopFcn = { @my_callback_fcn, 'My stop message'};
```

4 Specify the value of the `TimerFcn` callback. The example specifies the MATLAB commands in a text string:

```
t.TimerFcn = @(x,y)disp('Hello World!');
```

5 Start the timer object:

```
start(t)
```

The example outputs the following.

```
StartFcn event occurred at 10-Mar-2004 17:16:59
My start message
Hello World!
Hello World!
StopFcn event occurred at 10-Mar-2004 17:16:59
My stop message
```

6 Delete the timer object after you are finished with it.

```
delete(t)
```

Timer Object Execution Modes

In this section...
"Executing a Timer Callback Function Once" on page 7-19
"Executing a Timer Callback Function Multiple Times" on page 7-20
"Handling Callback Function Queuing Conflicts" on page 7-21

Executing a Timer Callback Function Once

The timer object supports several execution modes that determine how it schedules the timer callback function (TimerFcn) for execution. You specify the execution mode by setting the value of the ExecutionMode property.

To execute a timer callback function once, set the ExecutionMode property to 'singleShot'. This is the default execution mode. In this mode, the timer object starts the timer and, after the time period specified in the StartDelay property elapses, adds the timer callback function (TimerFcn) to the MATLAB execution queue. When the timer callback function finishes, the timer stops.

The following figure graphically illustrates the parts of timer callback execution for a singleShot execution mode. The shaded area in the figure, labelled queue lag, represents the indeterminate amount of time between when the timer adds a timer callback function to the MATLAB execution queue and when the function starts executing. The duration of this lag is dependent on what other processing MATLAB happens to be doing at the time.

singleShot

Timer Callback Execution (singleShot Execution Mode)

Executing a Timer Callback Function Multiple Times

The timer object supports three multiple-execution modes:

- `'fixedRate'`
- `'fixedDelay'`
- `'fixedSpacing'`

In many ways, these execution modes operate the same:

- The `TasksToExecute` property specifies the number of times you want the timer to execute the timer callback function (`TimerFcn`).
- The `Period` property specifies the amount of time between executions of the timer callback function.
- The `BusyMode` property specifies how the timer object handles queuing of the timer callback function when the previous execution of the callback function has not completed. See "Handling Callback Function Queuing Conflicts" on page 7-21 for more information.

The execution modes differ only in where they start measuring the time period between executions. The following table describes these differences.

Execution Mode	Description
`'fixedRate'`	Time period between executions begins immediately after the timer callback function is added to the MATLAB execution queue.
`'fixedDelay'`	Time period between executions begins when the timer function callback actually starts executing, after any time lag due to delays in the MATLAB execution queue.
`'fixedSpacing'`	Time period between executions begins when the timer callback function finishes executing.

The following figure illustrates the difference between these modes. Note that the amount of time between executions (specified by the `Period` property) remains the same. Only the point at which execution begins is different.

Differences Between Execution Modes

Handling Callback Function Queuing Conflicts

At busy times, in multiple-execution scenarios, the timer may need to add the timer callback function (TimerFcn) to the MATLAB execution queue before the previously queued execution of the callback function has completed. You can determine how the timer object handles this scenario by using the BusyMode property.

If you specify 'drop' as the value of the BusyMode property, the timer object skips the execution of the timer function callback if the previously scheduled callback function has not already completed.

If you specify 'queue', the timer object waits until the currently executing callback function finishes before queuing the next execution of the timer callback function.

> **Note** In `'queue'` mode, the timer object tries to make the average time between executions equal the amount of time specified in the `Period` property. If the timer object has to wait longer than the time specified in the `Period` property between executions of the timer function callback, it shortens the time period for subsequent executions to make up the time.

If the `BusyMode` property is set to `'error'`, the timer object stops and executes the timer object error callback function (`ErrorFcn`), if one is specified.

Deleting Timer Objects from Memory

In this section...

Deleting One or More Timer Objects

When you are finished with a timer object, delete it from memory using the
`delete` function:

```
delete(t)
```

When you delete a timer object, workspace variables that referenced the object
remain. Deleted timer objects are invalid and cannot be reused. Use the `clear`
command to remove workspace variables that reference deleted timer objects.

To remove all timer objects from memory, enter

```
delete(timerfind)
```

For information about the `timerfind` function, see "Finding Timer Objects in
Memory" on page 7-24.

Testing the Validity of a Timer Object

To test if a timer object has been deleted, use the `isvalid` function. The
`isvalid` function returns logical 0 (`false`) for deleted timer objects:

```
isvalid(t)
ans =

     0
```

Finding Timer Objects in Memory

In this section...
"Finding All Timer Objects" on page 7-24
"Finding Invisible Timer Objects" on page 7-24

Finding All Timer Objects

To find all the timer objects that exist in memory, use the `timerfind` function. This function returns an array of timer objects. If you leave off the semicolon, and there are multiple timer objects in the array, `timerfind` displays summary information in a table:

```
t1 = timer;
t2 = timer;
t3 = timer;
t_array = timerfind

Timer Object Array

   Index:  ExecutionMode:  Period:  TimerFcn:   Name:
   1       singleShot      1        ''          timer-3
   2       singleShot      1        ''          timer-4
   3       singleShot      1        ''          timer-5
```

Using `timerfind` to determine all the timer objects that exist in memory can be helpful when deleting timer objects.

Finding Invisible Timer Objects

If you set the value of a timer object's `ObjectVisibility` property to `'off'`, the timer object does not appear in listings of existing timer objects returned by `timerfind`. The `ObjectVisibility` property provides a way for application developers to prevent end-user access to the timer objects created by their application.

Objects that are not visible are still valid. If you have access to the object (for example, from within the file that created it), you can set its properties. To

retrieve a list of all the timer objects in memory, including invisible ones, use the `timerfindall` function.

Performance

- "Analyzing Your Program's Performance" on page 8-2
- "Techniques for Improving Performance" on page 8-4

Analyzing Your Program's Performance

In this section...
"Overview" on page 8-2
"The Profiler Utility" on page 8-2
"Stopwatch Timer Functions" on page 8-2

Overview

The MATLAB Profiler graphical user interface and the stopwatch timer functions enable you to get back information on how your program is performing and help you identify areas that need improvement. The Profiler can be more useful in measuring relative execution time and in identifying specific performance bottlenecks in your code, while the stopwatch functions tend to be more useful for providing absolute time measurements.

The Profiler Utility

A good first step to speeding up your programs is to find out where the bottlenecks are. This is where you need to concentrate your attention to optimize your code.

The MATLAB software provides the MATLAB Profiler, a graphical user interface that shows you where your program is spending its time during execution. Use the Profiler to help you determine where you can modify your code to make performance improvements.

To start the Profiler, type `profile viewer` or select **Desktop > Profiler** in the MATLAB Command Window. See Profiling for Improving Performance in the MATLAB Desktop Tools and Development Environment documentation, and the `profile` function reference page.

Stopwatch Timer Functions

If you just need to get an idea of how long your program (or a portion of it) takes to run, or to compare the speed of different implementations of a program, you can use the stopwatch timer functions, `tic` and `toc`. Invoking

tic starts the timer, and the first subsequent toc stops it and reports the time elapsed between the two.

Use tic and toc as shown here:

```
tic
    -- run the program section to be timed --
toc
```

Keep in mind that tic and toc measure overall elapsed time. Make sure that no other applications are running in the background on your system that could affect the timing of your MATLAB programs.

Measuring Smaller Programs

Shorter programs sometimes run too fast to get useful data from tic and toc. When this is the case, try measuring the program running repeatedly in a loop, and then average to find the time for a single run:

```
tic
    for k = 1:100
        -- run the program --
    end
toc
```

Using tic and toc Versus the cputime Function

Although it is possible to measure performance using the cputime function, it is recommended that you use the tic and toc functions for this purpose exclusively. It has been the general rule for CPU-intensive calculations run on Microsoft Windows machines that the elapsed time using cputime and the elapsed time using tic and toc are close in value, ignoring any first time costs. There are cases however that show a significant difference between these two methods. For example, in the case of a Pentium 4 with hyperthreading running Windows, there can be a significant difference between the values returned by cputime versus tic and toc.

Techniques for Improving Performance

Preallocating Arrays

`for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can adversely affect performance and memory use. Repeatedly resizing arrays often requires that MATLAB spend extra time looking for larger contiguous blocks of memory and then moving the array into those blocks. You can often improve on code execution time by preallocating the maximum amount of space that would be required for the array ahead of time.

The following code creates a scalar variable `x`, and then gradually increases the size of `x` in a `for` loop instead of preallocating the required amount of memory at the start:

```
x = 0;
for k = 2:1000
    x(k) = x(k-1) + 5;
end
```

Change the first line to preallocate a 1-by-1000 block of memory for `x` initialized to zero. This time there is no need to repeatedly reallocate memory and move data as more values are assigned to `x` in the loop:

```
x = zeros(1, 1000);
for k = 2:1000
   x(k) = x(k-1) + 5;
end
```

Preallocation Functions

Preallocation makes it unnecessary for MATLAB to resize an array each time you enlarge it. Use the appropriate preallocation function for the kind of array you are working with.

Array Type	Function	Examples
Numeric	`zeros`	`y = zeros(1, 100);`

Cell	`cell`	`B = cell(2, 3);`
		`B{1,3} = 1:3;`
		`B{2,2} = 'string';`

Preallocating a Nondouble Matrix

When you preallocate a block of memory to hold a matrix of some type other than `double`, avoid using the method

```
A = int8(zeros(100));
```

This statement preallocates a 100-by-100 matrix of `int8` first by creating a full matrix of `double`s, and then converting each element to `int8`. This costs time and uses memory unnecessarily.

The next statement shows how to do this more efficiently:

```
A = zeros(100, 'int8');
```

Limiting Size and Complexity

Running programs that are unusually large or complex can put a strain on your system's resources. For example, a program that nearly exceeds memory capacity may work some of the time and sometimes not, depending on the commands it uses and on what other applications are running at the time. An example of unnecessary complexity might be having a large number of `if` and `else` statements where `switch` and `case` might be more suitable. This can

also lead to performance and space problems. If you see the following error message displayed, this is likely to be the source of the problem:

```
The input was too complicated or too big for MATLAB to parse
```

If you have a program file that includes thousands of variables or functions, tens of thousands of statements, or hundreds of language keyword pairs (e.g., `if-else`, or `try-catch`), then making some of the changes suggested here is likely to not only boost its performance and reliability, but should make your program code easier to understand and maintain as well.

- Split large script files into smaller ones, having the first file call the second if necessary.

- Take your larger chunks of program code and make separate functions (or subfunctions and nested functions) of them.

- If you have functions or expressions by that seem overly complicated, make smaller and simpler functions or expressions of them. Simpler functions are also more likely to be made into utility functions that you can share with others.

Assigning to Variables

For best performance, keep the following suggestions in mind when assigning values to variables.

Changing a Variable's Data Type or Dimension

Changing the class or array shape of an existing variable slows MATLAB down as it must take extra time to process this. When you need to store data of a different type, it is advisable to create a new variable.

This code changes the type for X from `double` to `char`, which has a negative impact on performance:

```
X = 23;
    .
-- other code  --
    .
X = 'A';                  % X changed from type double to char
    .
```

```
-- other code  --
```

Assigning Real and Complex Numbers

Assigning a complex number to a variable that already holds a real number impacts the performance of your program. Similarly, you should not assign a real value to a variable that already holds a complex value.

Using Appropriate Logical Operators

When performing a logical AND or OR operation, you have a choice of two operators of each type.

Operator	Description
&, |	Perform logical AND and OR on arrays element by element
&&, ||	Perform logical AND and OR on scalar values with short-circuiting

In `if` and `while` statements, it is more efficient to use the short-circuiting operators, `&&` for logical AND and `||` for logical OR. This is because these operators often do not have to evaluate the entire logical expression. For example, MATLAB evaluates only the first part of this expression whenever the number of input arguments is less than three:

```
if (nargin >= 3) && (ischar(varargin{3}))
```

See Short-Circuit Operators in the MATLAB documentation for a discussion on short-circuiting with `&&` and `||`.

Overloading Built-In Functions

Overloading MATLAB built-in functions on any of the standard MATLAB data classes can negatively affect performance. For example, if you overload the `plus` function to handle any of the integer classes differently, you may hinder certain optimizations in the MATLAB built-in function code for `plus`, and thus may slow down any programs that make use of this overload.

Functions Are Generally Faster Than Scripts

Your code executes more quickly if it is implemented in a function rather than a script.

Load and Save Are Faster Than File I/O Functions

If you have a choice of whether to use `load` and `save` instead of the low-level MATLAB file I/O routines such as `fread` and `fwrite`, choose the former. `load` and `save` have been optimized to run faster and reduce memory fragmentation.

Vectorizing Loops

The MATLAB software uses a matrix language, which means it is designed for vector and matrix operations. You can often speed up your code by using vectorizing algorithms that take advantage of this design. *Vectorization* means converting `for` and `while` loops to equivalent vector or matrix operations.

Simple Example of Vectorizing

Here is one way to compute the sine of 1001 values ranging from 0 to 10:

```
i = 0;
for t = 0:.01:10
    i = i + 1;
    y(i) = sin(t);
end
```

A vectorized version of the same code is

```
t = 0:.01:10;
y = sin(t);
```

The second example executes much faster than the first and is the way MATLAB is meant to be used. Test this on your system by creating scripts that contain the code shown, and then using the `tic` and `toc` functions to measure the performance.

Advanced Example of Vectorizing

`repmat` is an example of a function that takes advantage of vectorization. It accepts three input arguments: an array A, a row dimension M, and a column dimension N.

`repmat` creates an output array that contains the elements of array A, replicated and "tiled" in an M-by-N arrangement:

```
A = [1 2 3; 4 5 6];

B = repmat(A,2,3);
B =
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
    1    2    3    1    2    3    1    2    3
    4    5    6    4    5    6    4    5    6
```

`repmat` uses vectorization to create the indices that place elements in the output array:

```
function B = repmat(A, M, N)

% Step 1 Get row and column sizes
[m,n] = size(A);

% Step 2 Generate vectors of indices from 1 to row/column size
mind = (1:m)';
nind = (1:n)';

% Step 3 Create index matrices from vectors above
mind = mind(:,ones(1, M));
nind = nind(:,ones(1, N));

% Step 4 Create output array
B = A(mind,nind);
```

Step 1, above, obtains the row and column sizes of the input array.

Step 2 creates two column vectors. `mind` contains the integers from 1 through the row size of `A`. The `nind` variable contains the integers from 1 through the column size of `A`.

Step 3 uses a MATLAB vectorization trick to replicate a single column of data through any number of columns. The code is

```
B = A(:,ones(1,nCols))
```

where `nCols` is the desired number of columns in the resulting matrix.

Step 4 uses array indexing to create the output array. Each element of the row index array, `mind`, is paired with each element of the column index array, `nind`, using the following procedure:

1 The first element of `mind`, the row index, is paired with each element of `nind`. MATLAB moves through the `nind` matrix in a columnwise fashion, so `mind(1,1)` goes with `nind(1,1)`, and then `nind(2,1)`, and so on. The result fills the first row of the output array.

2 Moving columnwise through `mind`, each element is paired with the elements of `nind` as above. Each complete pass through the `nind` matrix fills one row of the output array.

Caution   While `repmat` can take advantage of vectorization, it can do so at the expense of memory usage. When this is the case, you might find the `bsxfun` function be more appropriate in this respect.

Functions Used in Vectorizing

Some of the most commonly used functions for vectorizing are as follows

Function	Description
`all`	Test to determine if all elements are nonzero
`any`	Test for any nonzeros
`cumsum`	Find cumulative sum

Function	Description
diff	Find differences and approximate derivatives
find	Find indices and values of nonzero elements
ind2sub	Convert from linear index to subscripts
ipermute	Inverse permute dimensions of a multidimensional array
logical	Convert numeric values to logical
meshgrid	Generate X and Y arrays for 3-D plots
ndgrid	Generate arrays for multidimensional functions and interpolation
permute	Rearrange dimensions of a multidimensional array
prod	Find product of array elements
repmat	Replicate and tile an array
reshape	Change the shape of an array
shiftdim	Shift array dimensions
sort	Sort array elements in ascending or descending order
squeeze	Remove singleton dimensions from an array
sub2ind	Convert from subscripts to linear index
sum	Find the sum of array elements

Avoid Large Background Processes

Avoid running large processes in the background at the same time you are executing your program in MATLAB. This frees more CPU time for your MATLAB session.

9

Memory Usage

Memory Allocation

For more information on memory management, see Technical Note 1106: "Memory Management Guide" at the following URL:

`http://www.mathworks.com/support/tech-notes/1100/1106.html`

Memory Allocation for Arrays

The topics below provide information on how the MATLAB software allocates memory when working with arrays and variables. The purpose is to help you use memory more efficiently when writing code. Most of the time, however, you should not need to be concerned with these internal operations as MATLAB handles data storage for you automatically.

- "Creating and Modifying Arrays" on page 9-2
- "Copying Arrays" on page 9-3
- "Array Headers" on page 9-5
- "Function Arguments" on page 9-6

Note Any information on how the MATLAB software handles data internally is subject to change in future releases.

Creating and Modifying Arrays

When you assign a numeric or character array to a variable, MATLAB allocates a contiguous virtual block of memory and stores the array data in that block. MATLAB also stores information about the array data, such as its class and dimensions, in a separate, small block of memory called a *header*.

If you add new elements to an existing array, MATLAB expands the existing array in memory in a way that keeps its storage contiguous. This usually requires finding a new block of memory large enough to hold the expanded array. MATLAB then copies the contents of the array from its original location to this new block in memory, adds the new elements to the array in this block, and frees up the original array location in memory.

If you remove elements from an existing array, MATLAB keeps the memory storage contiguous by removing the deleted elements, and then compacting its storage in the original memory location.

Working with Large Data Sets. If you are working with large data sets, you need to be careful when increasing the size of an array to avoid getting errors caused by insufficient memory. If you expand the array beyond the available contiguous memory of its original location, MATLAB must make a copy of the array and set this copy to the new value. During this operation, there are two copies of the original array in memory. This temporarily doubles the amount of memory required for the array and increases the risk of your program running out of memory during execution. It is better to preallocate sufficient memory for the largest potential size of the array at the start. See "Preallocating Arrays" on page 8-4.

Copying Arrays

Internally, multiple variables can point to the same block of data, thus sharing that array's value. When you copy a variable to another variable (e.g., `B = A`), MATLAB makes a copy of the array reference, but not the array itself. As long as you do not modify the contents of the array, there is no need to store more than one copy of it. If you do modify any elements of the array, MATLAB makes a copy of the array and then modifies that copy.

The following example demonstrates this. Start by creating a simple script `memUsed.m` to display how much memory is currently being used by your MATLAB process. Put these two lines of code in the script:

```
[usr, sys] = memory;
usr.MemUsedMATLAB
```

Get an initial reading of how much memory is currently being used by your MATLAB process:

```
format short eng;
memUsed
ans =
   295.4977e+006
```

Create a 2000-by-2000 numeric array A. This uses about 32MB of memory:

```
A = magic(2000);
memUsed
ans =
   327.6349e+006
```

Make a copy of array A in B. As there is no need at this point to have two copies of the array data, MATLAB only makes a copy of the array reference. This requires no significant additional memory:

```
B = A;
memUsed
ans =
   327.6349e+006
```

Now modify B by making it one half its original size (i.e., set 1000 rows to empty). This requires that MATLAB make a copy of at least the first 1000 rows of the A array, and assign that copy to B:

```
B(1001:2000,:) = [];
format short;   size(B)
ans =
       1000        2000
```

Check the memory used again. Even though B is significantly smaller than it was originally, the amount of memory used by the MATLAB process has increased by about 16 MB (1/2 of the 32 MB originally required for A) because B could no longer remain as just a reference to A:

```
format short eng;   memUsed
ans =
   343.6421e+006
```

Array Headers

When you assign an array to a variable, MATLAB also stores information about the array (such as class and dimensions) in a separate piece of memory called a header. For most arrays, the memory required to store the header is insignificant. There is a small advantage to storing large data sets in a small number of large arrays as opposed to a large number of small arrays. This is because the former configuration requires fewer array headers.

Structure and Cell Arrays. For structures and cell arrays, MATLAB creates a header not only for each array, but also for each field of the structure and for each cell of a cell array. Because of this, the amount of memory required to store a structure or cell array depends not only on how much data it holds, but also on how it is constructed.

For example, take a scalar structure array S1 having fields R, G, and B. Each field of size 100-by-50 requires one array header to describe the overall structure, one header for each unique field name, and one header per field for the 1-by-1 structure array. This makes a total of seven array headers for the entire data structure:

```
S1.R(1:100,1:50)
S1.G(1:100,1:50)
S1.B(1:100,1:50)
```

On the other hand, take a 100-by-50 structure array S2 in which each element has scalar fields R, G, and B. You only need one array header to describe the overall structure, one for each unique field name, and one per field for each of the 5,000 elements of the structure, making a total of 15,004 array headers for the entire data structure:

```
S2(1:100,1:50).R
S2(1:100,1:50).G
S2(1:100,1:50).B
```

Even though S1 and S2 contain the same amount of data, S1 uses significantly less space in memory. Not only is less memory required, but there is a corresponding speed benefit to using the S1 format, as well.

See "Cell Arrays" and "Structures" under "Data Structures and Memory" on page 9-7.

Memory Usage Reported By the whos Function. The whos function displays the amount of memory consumed by any variable. For reasons of simplicity, whos reports only the memory used to store the actual data. It does not report storage for the array header, for example.

Function Arguments

MATLAB handles arguments passed in function calls in a similar way. When you pass a variable to a function, you are actually passing a reference to the data that the variable represents. As long as the input data is not modified by the function being called, the variable in the calling function and the variable in the called function point to the same location in memory. If the called function modifies the value of the input data, then MATLAB makes a copy of the original array in a new location in memory, updates that copy with the modified value, and points the input variable in the called function to this new array.

In the example below, function myfun modifies the value of the array passed into it. MATLAB makes a copy in memory of the array pointed to by A, sets variable X as a reference to this new array, and then sets one row of X to zero. The array referenced by A remains unchanged:

```
A = magic(500);
myfun(A);

function myfun(X)
X(400,:) = 0;
```

If the calling function needs the modified value of the array it passed to myfun, you need to return the updated array as an output of the called function, as shown here for variable A:

```
A = magic(500);
A = myfun(A);
sprintf('The new value of A is %d', A)

function Y = myfun(X)
X(400,:) = 0;
Y = X;
```

Data Structures and Memory

Memory requirements differ for the various types of MATLAB data structures. You may be able to reduce the amount of memory used for these structures by considering how MATLAB stores them.

Numeric Arrays

MATLAB requires 1, 2, 4, or 8 bytes to store 8-bit, 16-bit, 32-bit, and 64-bit signed and unsigned integers, respectively. For floating-point numbers, MATLAB uses 4 or 8 bytes for `single` and `double` types. To conserve memory when working with numeric arrays, MathWorks recommends that you use the smallest integer or floating-point type that will contain your data without overflowing. For more information, see "Numeric Types" in the MATLAB Programming Fundamentals documentation.

Complex Arrays

MATLAB stores complex data as separate real and imaginary parts. If you make a copy of a complex array variable, and then modify only the real or imaginary part of the array, MATLAB creates a new array containing both real and imaginary parts.

Sparse Matrices

It is best to store matrices with values that are mostly zero in sparse format. Sparse matrices can use less memory and may also be faster to manipulate than full matrices. You can convert a full matrix to sparse format using the `sparse` function.

Compare two 1000-by-1000 matrices: `X`, a matrix of doubles with 2/3 of its elements equal to zero; and `Y`, a sparse copy of `X`. The following example shows that the sparse matrix requires approximately half as much memory:

```
whos
  Name      Size                   Bytes  Class

  X      1000x1000               8000000  double array
  Y      1000x1000               4004000  double array (sparse)
```

Cell Arrays

In addition to data storage, cell arrays require a certain amount of additional memory to store information describing each cell. This information is recorded in a *header*, and there is one header for each cell of the array. You can determine the amount of memory required for a cell array header by finding the number of bytes consumed by a 1-by-1 cell that contains no data, as shown below for a 32-bit system:

```
A = {[]};       % Empty cell array

whos A
  Name        Size              Bytes  Class     Attributes

  A           1x1                  60  cell
```

In this case, MATLAB shows the number of bytes required for each header in the cell array on a 32-bit system to be 60. This is the header size that is used in all of the 32-bit examples in this section. For 64-bit systems, the header size is assumed to be 112 bytes in this documentation. You can find the correct header size on a 64-bit system using the method just shown for 32 bits.

To predict the size of an entire cell array, multiply the number you have just derived for the header by the total number of cells in the array, and then add to that the number of bytes required for the data you intend to store in the array:

```
(header_size x number_of_cells) + data
```

So a 10-by-20 cell array that contains 400 bytes of data would require 22,800 bytes of memory on a 64-bit system:

```
(112 x 200) + 400 = 22800
```

Note While numeric arrays must be stored in contiguous memory, structures and cell arrays do not.

Example 1 – Memory Allocation for a Cell Array. The following 4-by-1 cell array records the brand name, screen size, price, and on-sale status for three laptop computers:

```
Laptops = {['SuperrrFast 89X', 'ReliablePlus G5', ...
            'UCanA4dIt 140L6']; ...
           [single(17), single(15.4), single(14.1)]; ...
           [2499.99, 1199.99, 499.99]; ...
           [true, true, false]};
```

On a 32-bit system, the cell array header alone requires 60 bytes per cell:

```
4 cells * 60 bytes per cell = 240 bytes for the cell array
```

Calculate the memory required to contain the data in each of the four cells:

```
45 characters * 2 bytes per char = 90 bytes
 3 doubles * 8 bytes per double = 24 bytes
 3 singles * 4 bytes per single = 12 bytes
 3 logicals * 1 byte per logical = 3 bytes

90 + 24 + 12 + 3 = 129 bytes for the data
```

Add the two, and then compare your result with the size returned by MATLAB:

```
240 + 129 = 369 bytes total

whos Laptops
  Name          Size          Bytes  Class     Attributes

  Laptops       4x1             369  cell
```

Structures

```
S.A = [];
B = whos('S');
B.bytes - 60
ans =
    64
```

Compute the memory needed for a structure array as follows:

```
32-bit systems:  fields x ((60 x array elements) + 64) + data
64-bit systems:  fields x ((112 x array elements) + 64) + data
```

On a 64-bit computer system, a 4-by-5 structure `Clients` with fields `Address` and `Phone` uses 4,608 bytes just for the structure:

```
2 fields x ((112 x 20) + 64) = 2 x (2240 + 64) = 4608 bytes
```

To that sum, you must add the memory required to hold the data assigned to each field. If you assign a 25-character string to `Address` and a 12-character string to `Phone` in each element of the 4-by-5 `Clients` array, you use 1480 bytes for data:

```
(25+12) characters * 2 bytes per char * 20 elements = 1480 bytes
```

Add the two and you see that the entire structure consumes 6,088 bytes of memory.

Example 1 – Memory Allocation for a Structure Array. Compute the amount of memory that would be required to store the following 6-by-5 structure array having the following four fields on a 32-bit system:

```
A:  5-by-8-by-6 signed 8-bit integer array
B:  1-by-200 single array
C:  30-by-30 unsigned 16-bit integer array
D:  1-by-27 character array
```

Construct the array:

```
A = int8(ones(5,8,6));
B = single(1:500);
C = uint16(magic(30));
D = 'Company Name: MathWorks';

s = struct('f1', A, 'f2', B, 'f3', C, 'f4', D);

for m=1:6
   for n=1:5
      s(m,n)=s(1,1);
   end
end
```

Calculate the amount of memory required for the structure itself, and then for the data it contains:

```
structure = fields x ((60 x array elements) + 64) =
            4 x ((60 x 30) + 64) = 7,456 bytes

data = (field1 + field2 + field3 + field4) x array elements =
       (240 + 2000 + 1800 + 54) x 30 = 122,820 bytes
```

Add the two, and then compare your result with the size returned by MATLAB:

```
Total bytes calculated for structure s: 7,456 + 122,820 = 130,276

whos s
  Name      Size            Bytes  Class     Attributes

  s         6x5             130036  struct
```

Memory Management Functions

The following functions can help you to manage memory use while running the MATLAB software:

- `memory` displays or returns information about how much memory is available and how much is used by MATLAB. This includes the following:

    - Size of the largest single array MATLAB can create at this time.

    - Total size of the virtual address space available for data.

    - Total amount of memory used by the MATLAB process for both libraries and data.

    - Available and total Virtual Memory for the MATLAB software process.

    - Available system memory, including both physical memory and paging file.

    - Available and the total physical memory (RAM) of the computer.

- `whos` shows how much memory MATLAB currently has allocated for variables in the workspace.

- `pack` saves existing variables to disk, and then reloads them contiguously. This reduces the chances of running into problems due to memory fragmentation.

- `clear` removes variables from memory. One way to increase the amount of available memory is to periodically clear variables from memory that you no longer need.

    If you use `pack` and there is still not enough free memory to proceed, you probably need to remove some of the variables you are no longer using from memory. Use `clear` to do this.

- `save` selectively stores variables to the disk. This is a useful technique when you are working with large amounts of data. Save data to the disk periodically, and then use the `clear` function to remove the saved data from memory.

- `load` reloads a data file saved with the `save` function.

- `quit` exits MATLAB and returns all allocated memory to the system. This can be useful on The Open Group UNIX systems, which do not free up

memory allocated to an application (e.g., MATLAB) until the application exits.

You can use the `save` and `load` functions in conjunction with the `quit` command to free memory by:

1 Saving any needed variables with the `save` function.

2 Quitting MATLAB to free all memory allocated to MATLAB.

3 Starting a new MATLAB session and loading the saved variables back into the clean MATLAB workspace.

The whos Function

The whos command can give you an idea of the memory used by MATLAB variables.

```
A = ones(10,10);
whos
   Name        Size              Bytes  Class      Attributes
   A          10x10                800  double
```

Note that whos does not include information about

- Memory used by MATLAB (e.g., Sun Java code and plots).

- Memory used for most objects (e.g., time series, custom) .

- Memory for variables not in the calling workspace .

- Shared data copies. `whos` shows bytes used for a shared data copy even when it does not use any memory. This example shows that `whos` reports an array (`A`) and a shared data copy of that array (`B`) separately, even though the data exists only once in memory:

Store 400 MB array as `A`. Memory used = 381MB (715 MB – 334 MB) :

```
memory
Memory used by MATLAB:              334 MB (3.502e+008 bytes)

A = rand(5e7,1);
```

```
memory
Memory used by MATLAB:                715 MB (7.502e+008 bytes)

whos
  Name            Size              Bytes  Class      Attributes

  A          50000000x1          400000000  double
```

Create B and point it to A. Note that although whos shows both A and B, there is only one copy of the data in memory. No additional memory is consumed by assigning A to B:

```
B = A;

memory
Memory used by MATLAB:                715 MB (7.502e+008 bytes)

whos
  Name            Size              Bytes  Class      Attributes

  A          50000000x1          400000000  double
  B          50000000x1          400000000  double
```

Modifying B(1) copies all of A to B and changes the value of B(1). Memory used = 382MB (1097 MB – 715 MB). Now there are two copies of the data in memory, yet the output of whos does not change:

```
B(1) = 3;

memory
Memory used by MATLAB:               1097 MB (1.150e+009 bytes)

whos
  Name            Size              Bytes  Class      Attributes

  A          50000000x1          400000000  double
  B          50000000x1          400000000  double
```

Strategies for Efficient Use of Memory

In this section...
"Ways to Reduce the Amount of Memory Required" on page 9-15
"Using Appropriate Data Storage" on page 9-17
"How to Avoid Fragmenting Memory" on page 9-20
"Reclaiming Used Memory" on page 9-22

Ways to Reduce the Amount of Memory Required

The source of many "out of memory" problems often involves analyzing or processing an existing large set of data such as in a file or a database. This requires bringing all or part of the data set into the MATLAB software process. The following techniques deal with minimizing the required memory during this stage.

Load Only As Much Data As You Need

Only import into MATLAB as much of a large data set as you need for the problem you are trying to solve. This is not usually a problem when importing from sources, such as a database where you can explicitly search for elements matching a query. But this is a common problem with loading large flat text or binary files. Many users are tempted to try and load the entire file first, and then process it with MATLAB. Be sure to use the appropriate MATLAB function to load parts of files.

Text Files. Use the `textscan` function to access parts of a large text file by reading only the selected columns and rows. If you specify the number of rows or a repeat format number with `textscan`, MATLAB calculates the exact amount of memory required beforehand.

Binary Files. You can use low-level binary file I/O functions, such as `fread`, to access parts of any file that has a known format. For binary files of an unknown format, try using memory mapping with the `memmapfile` function.

MAT-Files. Use the `whos` function with the `-file` option to preview the file. This command displays each array in the MAT-file that you specify and the number of bytes in the array:

```
whos -file session1.mat
  Name        Size              Bytes  Class     Attributes

  S2          1x1                 723  struct
  x           100x200              72  double    sparse
  Mat4        4x20                640  double
  A           3151872x1       3151872  uint8
  Seq         1x912211         912211  int8
```

If there are large arrays in the MAT-file that you do not need for your current task, you can selectively import only those variables that you want using `load`.

HDF Files. You can load parts of HDF files using the `hdfread` and `hdf5read` functions.

Image, Audio, and Video Files. The MATLAB functions that support loading from these types of files usually require that you import the entire file. To import portions of the file, use low-level I/O functions such as `fread`.

Process Data By Blocks

Consider block processing, that is, processing a large data set one section at a time in a loop. Reducing the size of the largest array in a data set reduces the size of any copies or temporaries needed. You can use this technique in either of two ways:

- For a subset of applications that you can break into separate chunks and process independently.

- For applications that only rely on the state of a previous block, such as filtering.

Avoid Creating Temporary Arrays

Avoid creating large temporary variables, and also make it a practice to clear those temporary variables you do use when they are no longer needed. For example, when you create a large array of zeros, instead of saving to a temporary variable A, and then converting A to a single:

```
A = zeros(1e6,1);
As = single(A);
```

use just the one command to do both operations:

```
A = zeros(1e6,1,'single');
```

Using the `repmat` function, array preallocation and `for` loops are other ways to work on `nondouble` data without requiring temporary storage in memory.

Use Nested Functions to Pass Fewer Arguments

When working with large data sets, be aware that MATLAB makes a temporary copy of an input variable if the called function modifies its value. This temporarily doubles the memory required to store the array, which causes MATLAB to generate an error if sufficient memory is not available.

One way to use less memory in this situation is to use nested functions. A nested function shares the workspace of all outer functions, giving the nested function access to data outside of its usual scope. In the example shown here, nested function `setrowval` has direct access to the workspace of the outer function `myfun`, making it unnecessary to pass a copy of the variable in the function call. When `setrowval` modifies the value of A, it modifies it in the workspace of the calling function. There is no need to use additional memory to hold a separate array for the function being called, and there also is no need to return the modified value of A:

```
function myfun
A = magic(500);

    function setrowval(row, value)
    A(row,:) = value;
    end

setrowval(400, 0);
disp('The new value of A(399:401,1:10) is')
A(399:401,1:10)
end
```

Using Appropriate Data Storage

MATLAB provides you with different sizes of data classes, such as `double` and `uint8`, so you do not need to use large classes to store your smaller segments

of data. For example, it takes 7,000 KB less memory to store 1,000 small unsigned integer values using the `uint8` class than it does with `double`.

Use the Appropriate Numeric Class

The numeric class you should use in MATLAB depends on your intended actions. The default class `double` gives the best precision, but requires 8 bytes per element of memory to store. If you intend to perform complicated math such as linear algebra, you must use a floating-point class such as a `double` or `single`. The `single` class requires only 4 bytes. There are some limitations on what you can do with `singles`, but most MATLAB Math operations are supported.

If you just need to carry out simple arithmetic and you represent the original data as integers, you can use the integer classes in MATLAB. The following is a list of numeric classes, memory requirements (in bytes), and the supported operations.

Class (Data Type)	Bytes	Supported Operations
`single`	4	Most math
`double`	8	All math
`logical`	1	Logical/conditional operations
`int8, uint8`	1	Arithmetic and some simple functions
`int16, uint16`	2	Arithmetic and some simple functions
`int32, uint32`	4	Arithmetic and some simple functions
`int64, int64`	8	Arithmetic and some simple functions

Reduce the Amount of Overhead When Storing Data

MATLAB arrays (implemented internally as `mxArrays`) require room to store meta information about the data in memory, such as type, dimensions, and attributes. This takes about 80 bytes per array. This overhead only becomes an issue when you have a large number (e.g., hundreds or thousands) of small `mxArrays` (e.g., scalars). The `whos` command lists the memory used by variables, but does not include this overhead.

Because simple numeric arrays (comprising one mxArray) have the least overhead, you should use them wherever possible. When data is too complex to store in a simple array (or matrix), you can use other data structures.

Cell arrays are comprised of separate mxArrays for each element. As a result, cell arrays with many small elements have a large overhead.

Structures require a similar amount of overhead per field (see the documentation on "Array Headers" on page 9-5 above). Structures with many fields and small contents have a large overhead and should be avoided. A large array of structures with numeric scalar fields requires much more memory than a structure with fields containing large numeric arrays.

Also note that while MATLAB stores numeric arrays in contiguous memory, this is not the case for structures and cell arrays.

Import Data to the Appropriate MATLAB Class

When reading data from a binary file with fread, it is a common error to specify only the class of the data in the file, and not the class of the data MATLAB uses once it is in the workspace. As a result, the default double is used even if you are reading only 8-bit values. For example,

```
fid = fopen('large_file_of_uint8s.bin', 'r');
a = fread(fid, 1e3, 'uint8');              % Requires 8k
whos a
  Name        Size              Bytes  Class     Attributes

  a           1000x1             8000  double


a = fread(fid, 1e3, 'uint8=>uint8');       % Requires 1k
whos a
  Name        Size              Bytes  Class     Attributes

  a           1000x1             1000  uint8
```

Make Arrays Sparse When Possible

If your data contains many zeros, consider using sparse arrays, which store only nonzero elements. The example below compares the space required for storage of an array of mainly zeros:

```
A = diag(1e3,1e3);    % Full matrix with ones on the diagonal
As = sparse(A)        % Sparse matrix with only nonzero elements
whos
  Name      Size                    Bytes  Class

  A       1001x1001                8016008  double array
  As      1001x1001                   4020  double array (sparse)
```

You can see that this array requires only approximately 4 KB to be stored as sparse, but approximately 8 MB as a full matrix. In general, for a sparse double array with `nnz` nonzero elements and `ncol` columns, the memory required is

- 16 * nnz + 8 * ncol + 8 bytes (on a 64 bit machine)

- 12 * nnz + 4 * ncol + 4 bytes (on a 32 bit machine)

Note that MATLAB does not support all mathematical operations on sparse arrays.

How to Avoid Fragmenting Memory

MATLAB always uses a contiguous segment of memory to store a numeric array. As you manipulate this data, however, the contiguous block can become fragmented. When memory is fragmented, there may be plenty of free space, but not enough contiguous memory to store a new large variable. Increasing fragmentation can use significantly more memory than is necessary.

Preallocate Contiguous Memory When Creating Arrays

In the course of a MATLAB session, memory can become fragmented due to dynamic memory allocation and deallocation. `for` and `while` loops that incrementally increase, or *grow*, the size of a data structure each time through the loop can add to this fragmentation as they have to repeatedly find and allocate larger blocks of memory to store the data.

To make more efficient use of your memory, preallocate a block of memory large enough to hold the matrix at its final size before entering the loop. When you preallocate memory for an array, MATLAB reserves sufficient contiguous space for the entire full-size array at the beginning of the computation. Once

you have this space, you can add elements to the array without having to continually allocate new space for it in memory.

For more information on preallocation, see "Preallocating Arrays" on page 8-4.

Allocate Your Larger Arrays First

MATLAB uses a heap method of memory management. It requests memory from the operating system when there is not enough memory available in the heap to store the current variables. It reuses memory as long as the size of the memory segment required is available in the heap.

The following statements can require approximately 4.3 MB of RAM. This is because MATLAB may not be able to reuse the space previously occupied by two 1 MB arrays when allocating space for a 2.3 MB array:

```
a = rand(1e6,1);
b = rand(1e6,1);
clear
c = rand(2.3e6,1);
```

The simplest way to prevent overallocation of memory is to allocate the largest vectors first. These statements require only about 2.0 MB of RAM:

```
c = rand(2.3e6,1);
clear
a = rand(1e6,1);
b = rand(1e6,1);
```

Long-Term Usage (Windows Systems Only)

On 32-bit Microsoft Windows, the workspace of MATLAB can fragment over time due to the fact that the Windows memory manager does not return blocks of certain types and sizes to the operating system. Clearing the MATLAB workspace does not fix this problem. You can minimize the problem by allocating the largest variables first. This cannot address, however, the eventual fragmentation of the workspace that occurs from continual use of MATLAB over many days and weeks, for example. The only solution to this is to save your work and restart MATLAB.

The pack command, which saves all variables to disk and loads them back, does not help with this situation.

Reclaiming Used Memory

One simple way to increase the amount of memory you have available is to clear large arrays that you no longer use.

Save Your Large Data Periodically to Disk

If your program generates very large amounts of data, consider writing the data to disk periodically. After saving that portion of the data, use the clear function to remove the variable from memory and continue with the data generation.

Clear Old Variables from Memory When No Longer Needed

When you are working with a very large data set repeatedly or interactively, clear the old variable first to make space for the new variable. Otherwise, MATLAB requires temporary storage of equal size before overriding the variable. For example,

```
a = rand(100e6,1)              % 800 MB array
a = rand(100e6,1)              % New 800 MB array
??? Error using ==> rand
Out of memory. Type HELP MEMORY for your options.

clear a
a = rand(100e6,1)              % New 800 MB array
```

Resolving "Out of Memory" Errors

General Suggestions for Reclaiming Memory

The MATLAB software generates an `Out of Memory` message whenever it requests a segment of memory from the operating system that is larger than what is currently available. When you see the `Out of Memory` message, use any of the techniques discussed under "Strategies for Efficient Use of Memory" on page 9-15 to help optimize the available memory. If the `Out of Memory` message still appears, you can try any of the following:

- Compress data to reduce memory fragmentation.

- If possible, break large matrices into several smaller matrices so that less memory is used at any one time.

- If possible, reduce the size of your data.

- Make sure that there are no external constraints on the memory accessible to MATLAB. (On The Open Group UNIX[2] systems, use the `limit` command to check).

- Increase the size of the swap file. We recommend that you configure your system with twice as much swap space as you have RAM. See "Increasing System Swap Space" on page 9-26, below.

- Add more memory to the system.

---

2. UNIX is a registered trademark of The Open Group in the United States and other countries.

Setting the Process Limit

The platforms and operating systems that MATLAB supports have different memory characteristics and limitations. In particular, the *process limit* is the maximum amount of virtual memory a single process (or application) can address. On 32-bit systems, this is the most important factor limiting data set size. The process limit must be large enough for MATLAB to store all of the data it is to process, any MATLAB program files, the MATLAB executable itself, and additional state information.

Where possible, choose an operating system that maximizes this number, that is, a 64-bit operating system. The following is a list of MATLAB supported operating systems and their process limits.

Operating System	Process Limit
32-bit Microsoft Windows XP, Windows Vista™.	2 GB
32-bit Windows XP with 3 GB `boot.ini` switch or 32-bit Windows Vista with `increaseuserva` set (see later)	3 GB
32-bit Linux® (Linux is a registered trademark of Linus Torvalds.)	~3 GB
64-bit Windows XP, Apple Macintosh® OS X, Linux, or Sun Solaris™ running 32-bit MATLAB	≤ 4 GB
64-bit Windows XP, Windows Vista, Linux, or Solaris running 64-bit MATLAB	8 TB

To verify the current process limit of MATLAB on Windows systems, use the `memory` function.

```
Maximum possible array:              583 MB (6.111e+008 bytes) *
Memory available for all arrays:    1515 MB (1.588e+009 bytes) **
Memory used by MATLAB:               386 MB (4.050e+008 bytes)
Physical Memory (RAM):              2014 MB (2.112e+009 bytes)
```

```
   *  Limited by contiguous virtual address space available.
   ** Limited by virtual address space available.
```

When called with one output variable, the memory function returns or displays the following values. See the function reference for memory to find out how to use it with more than one output.

memory Return Value	Description
MaxPossibleArrayBytes	Size of the largest single array MATLAB can currently create
MemAvailableAllArrays	Total size of the virtual address space available for data
MemUsedMATLAB	Total amount of memory used by the MATLAB process

View the value against the Total entry in the Virtual Memory section. It is shown as 2 GB in the table, which is the default on Windows XP systems. On UNIX systems, see the ulimit command to view and set user limits including virtual memory.

Disabling Java VM on Startup

On UNIX systems, you can increase the workspace size by approximately 400 MB if you start MATLAB without the Sun Java VM. To do this, use the command line option -nojvm to start MATLAB. This also increases the size of the largest contiguous block (and therefore the largest matrix) by about the same.

Using -nojvm comes with a penalty in that you will lose many features that rely on the Java software, including the entire development environment.

Starting MATLAB with the -nodesktop option does not save any substantial amount of memory.

Shutting down other applications and services (e.g., using msconfig on Windows systems) can help if total system memory is the limiting factor, but usually process limit (on 32-bit machines) is the main limiting factor.

Increasing System Swap Space

The total memory available to applications on your computer is comprised of physical memory (RAM), plus a *page file*, or *swap file*, on disk. The page or swap file can be very large, even on 32-bit systems (e.g., 16 TB (terabytes) on 32-bit Windows, 512 TB on 64-bit Windows). The operating system allocates the virtual memory of each process to either physical RAM or to this file, depending on its needs and those of other processes.

How you set the swap space for your computer depends on what operating system you are running on.

UNIX Systems

For more information about swap space, type `pstat -s` at the UNIX command prompt. For detailed information on changing swap space, ask your system administrator.

Linux Systems

You can change your swap space by using the `mkswap` and `swapon` commands. For more information on the above commands, type `man` followed by the command name at the Linux prompt.

Windows XP Systems

Follow the steps shown here:

1 Right-click the `My Computer` icon, and select **Properties**.

2 In the `System Properties` GUI, select the **Advanced** tab. In the section labeled **Performance**, click the **Settings** button.

3 In the `Performance Options` GUI, click the **Advanced** tab. In the section labeled **Virtual Memory**, click the **Change** button

4 In the `Virtual Memory` GUI, under **Paging file size for selected drive**, you can change the amount of virtual memory.

Using the 3GB Switch on Windows Systems

Microsoft Windows XP systems can allocate 3 GB (instead of the default 2 GB) to processes, if you set an appropriate switch in the `boot.ini` file of the system. MathWorks recommends that you only do this with Windows XP SP2 systems or later. This gives an extra 1 GB of virtual memory to MATLAB, not contiguous with the rest of the memory. This enables you to store more data, but not larger arrays, as these are limited by contiguous space. This is mostly beneficial if you have enough RAM (e.g., 3 or 4 GB) to use it.

After setting the switch, confirm the new value of the virtual memory after restarting your computer and using the `memory` function.

```
[userview systemview] = memory;

systemview.VirtualAddressSpace
ans =
  Available: 1.6727e+009   % Virtual memory available to MATLAB.
      Total: 2.1474e+009   % Total virtual memory
```

For more documentation on this option, use the following URL:

http://support.microsoft.com/support/kb/articles/Q291/9/88.ASP

Similarly, on machines running Microsoft Windows Vista, you can achieve the same effect by using the command:

```
BCDEdit /set increaseuserva 3072
```

For more documentation on this option, use the following URL:

http://msdn2.microsoft.com/en-us/library/aa906211.aspx

Freeing Up System Resources on Windows Systems

There are no functions implemented to manipulate the way MATLAB handles Microsoft Windows system resources. Windows systems use these resources to track fonts, windows, and screen objects. Resources can be depleted by using multiple figure windows, multiple fonts, or several UI controls. One way to free up system resources is to close all inactive windows. Windows system icons still use resources.

10

Programming Tips

Introduction

This section is a categorized compilation of tips for the MATLAB programmer. Each item is relatively brief to help you browse through them and find information that is useful. Many of the tips include a reference to specific MATLAB documentation that gives you more complete coverage of the topic. You can find information on the following topics:

For suggestions on how to improve the performance of your MATLAB programs, and how to write programs that use memory more efficiently, see Improving Performance and Memory Usage

Command and Function Syntax

In this section...

Syntax Help

For help about the general syntax of MATLAB functions and commands, type

```
help syntax
```

Command and Function Syntaxes

You can enter MATLAB commands using either a *command* or *function* syntax. It is important to learn the restrictions and interpretation rules for both.

```
functionname arg1 arg2 arg3                % Command syntax
functionname('arg1','arg2','arg3')         % Function syntax
```

For more information: See Calling Functions in the MATLAB Programming Fundamentals documentation.

Command Line Continuation

You can continue most statements to one or more additional lines by terminating each incomplete line with an ellipsis (...). Breaking down a statement into a number of lines can sometimes result in a clearer programming style.

```
sprintf ('Example %d shows a command coded on %d lines.\n', ...
          exampleNumber, ...
```

```
            numberOfLines)
```

Note that you cannot continue an incomplete string to another line.

```
disp 'This statement attempts to continue a string ...
      to another line, resulting in an error.'
```

For more information: See Entering Long Statements in the MATLAB Desktop Tools and Development Environment documentation.

Completing Commands Using the Tab Key

You can save some typing when entering commands by entering only the first few letters of the command, variable, property, etc. followed by the **Tab** key. Typing the second line below (with **T** representing **Tab**) yields the expanded, full command shown in the third line:

```
f = figure;
set(f, 'papTuT,'cT)                      % Type this line.
set(f, 'paperunits','centimeters')    % This is what you get.
```

If there are too many matches for the string you are trying to complete, you will get no response from the first **Tab**. Press **Tab** again to see all possible choices:

```
set(f, 'paTT
PaperOrientation    PaperPositionMode  PaperType       Parent
PaperPosition       PaperSize          PaperUnits
```

For more information: See Tab Completion in the Command Window in the MATLAB Desktop Tools and Development Environment documentation

Recalling Commands

Use any of the following methods to simplify recalling previous commands to the screen:

• To recall an earlier command to the screen, press the up arrow key one or more times, until you see the command you want. If you want to modify the recalled command, you can edit its text before pressing **Enter** or **Return** to execute it.

- To recall a specific command by name without having to scroll through your earlier commands one by one, type the starting letters of the command, followed by the up arrow key.

- Open the Command History window (**Desktop > Command History**) to see all previous commands. Double-click the command you want to execute.

For more information: See Recalling Previous Lines and Command History Window in the MATLAB Desktop Tools and Development Environment documentation.

Clearing Commands

If you have typed a command that you then decide not to execute, you can clear it from the Command Window by pressing the Escape (**Esc**) key.

Suppressing Output to the Screen

To suppress output to the screen, end statements with a semicolon. This can be particularly useful when generating large matrices.

```
A = magic(100);     % Create matrix A, but do not display it.
```

Help

In this section...

Using the Help Browser

Open the Help browser from the MATLAB Command Window using one of the following:

- Click the question mark symbol in the toolbar.
- Select **Help > Product Help** from the menu.
- Type the word doc at the command prompt.

For more information: See Finding Information with the Help Browser in the MATLAB Desktop Tools and Development Environment documentation.

Help on Functions from the Help Browser

You can find help on a MATLAB function in any of the following ways:

- Click the ⊞ *fx* Functions button in the left pane of the Help browser. This brings you to that part of the Function Reference documentation that is organized by category. To use an alphabetical list to get help on a specific function, click Alphabetical List at the top of that page.

- Click the MATLAB button in the left pane of the Help browser. Look in the upper left corner of the page for links to either Functions: By Category, or Functions: Alphabetical List and click there for the type of documentation access you prefer.

- Type doc functionname at the command line.

Help on Functions from the Command Window

Several types of help on functions are available from the Command Window:

- To list all categories that you can request help on from the Command Window, just type

      help

- To see a list of functions for one of these categories, along with a brief description of each function, type help *category*. For example,

      help datafun

- To get help on a particular function, type help functionname. For example,

      help sortrows

Topical Help

In addition to the help on individual functions, you can get help on any of the following topics by typing help topicname at the command line.

Topic Name	Description
arith	Arithmetic operators
relop	Relational and logical operators
punct	Special character operators
slash	Arithmetic division operators
paren	Parentheses, braces, and bracket operators
precedence	Operator precedence

Topic Name	Description
datatypes	MATLAB classes, their associated functions, and operators that you can overload
lists	Comma separated lists
strings	Character strings
function_handle	Function handles and the @ operator
debug	Debugging functions
java	Using Sun Java from within the MATLAB software.
fileformats	A list of readable file formats
changeNotification	Microsoft Windows change notification

Paged Output

Before displaying a lengthy section of help text or code, put MATLAB into its paged output mode by typing more on. This breaks up any ensuing display into pages for easier viewing. Turn off paged output with more off.

Page through the displayed text using the space bar key. Or step through line by line using **Enter** or **Return**. Discontinue the display by pressing the **Q** key or **Ctrl+C**.

Writing Your Own Help

Start each program you write with a section of text providing help on how and when to use the function. If formatted properly, the MATLAB help function displays this text when you enter

```
help functionname
```

MATLAB considers the first group of consecutive lines immediately following the function definition line that begin with % to be the help section for the function. The first line without % as the left-most character ends the help.

For more information: See Help Text in the MATLAB Desktop Tools and Development Environment documentation.

Help for Subfunctions and Private Functions

You can write help for subfunctions using the same rules that apply to main functions. To display the help for the subfunction `mysubfun` in file `myfun.m`, type

```
help myfun>mysubfun
```

To display the help for a private function, precede the function name with `private/`. To get help on private function `myprivfun`, type

```
help private/myprivfun
```

Help for Methods and Overloaded Functions

You can write help text for object-oriented class methods implemented as MATLAB functions. Display help for the method by typing

```
help classname/methodname
```

where the file `methodname.m` resides in subfolder `@classname`.

For example, if you write a `plot` method for a class named `polynom`, (where the `plot` method is defined in the file `@polynom/plot.m`), you can display this help by typing

```
help polynom/plot
```

You can get help on overloaded MATLAB functions in the same way. To display the help text for the `eq` function as implemented in `matlab/iofun/@serial`, type

```
help serial/eq
```

Development Environment

In this section...
"Workspace Browser" on page 10-10
"Using the Find and Replace Utility" on page 10-10
"Commenting Out a Block of Code" on page 10-11
"Creating Functions from Command History" on page 10-11
"Editing Functions in EMACS" on page 10-11

Workspace Browser

The Workspace browser is a graphical interface to the variables stored in the MATLAB base and function workspaces. You can view, modify, save, load, and create graphics from workspace data using the browser. Select **Desktop > Workspace** to open the browser.

To view function workspaces, you need to be in debug mode.

For more information: See MATLAB Workspace in the MATLAB Desktop Tools and Development Environment documentation.

Using the Find and Replace Utility

Find any word or phrase in a group of files using the Find and Replace

utility. Click **Desktop > Current Folder**, click the ⚙️▾ icon at the top of the **Current Folder** window, and then select **Find Files** from the menu that appears.

When entering search text, you do not need to put quotes around a phrase. In fact, parts of words, like win for windows, will not be found if enclosed in quotes.

For more information: See Finding and Replacing Text in the Current File in the MATLAB Desktop Tools and Development Environment documentation.

Commenting Out a Block of Code

To comment out a block of text or code within the MATLAB editor,

1 Highlight the block of text you would like to comment out.

2 Holding the mouse over the highlighted text, select **Text > Comment** (or **Uncomment**, to do the reverse) from the toolbar. (You can also get these options by right-clicking the mouse.)

For more information: See Adding Comments in the MATLAB Desktop Tools and Development Environment documentation.

Creating Functions from Command History

If there is part of your current MATLAB session that you would like to add to a function, this is easily done using the Command History window:

1 Open this window by selecting **Desktop > Command History**.

2 Use **Shift+Click** or **Ctrl+Click** to select the lines you want to use. MATLAB highlights the selected lines.

3 Right-click once, and select **Create Script** from the menu that appears. MATLAB creates a new Editor window displaying the selected code.

Editing Functions in EMACS

If you use Emacs, you can download editing modes for editing MATLAB functions with GNU-Emacs or with early versions of Emacs from the MATLAB Central Web site:

```
http://www.mathworks.com/matlabcentral/
```

At this Web site, select **File Exchange**, and then **Utilities > Emacs**.

For more information: See General Preferences for the Editor/Debugger in the MATLAB Desktop Tools and Development Environment documentation.

Functions

In this section...

Function Structure

An MATLAB function consists of the components shown here:

```
function [x, y] = myfun(a, b, c)   % Function definition line
% H1 line -- A one-line summary of the function's purpose.
% Help text -- One or more lines of help text that explain
%   how to use the function. This text is displayed when
%   the user types "help functionname".

% The Function body normally starts after the first blank line.
% Comments -- Description (for internal use) of what the
%   function does, what inputs are expected, what outputs
%   are generated. Typing "help functionname" does not display
%   this text.

x = prod(a, b);                    % Start of Function code
```

For more information: See Basic Parts of a Function in the MATLAB Programming Fundamentals documentation.

Using Lowercase for Function Names

Function names appear in uppercase in MATLAB help text only to make the help easier to read. In practice, however, it is usually best to use lowercase when calling functions.

Case requirements depend on the case sensitivity of the operating system you are using. As a rule, naming and calling functions using lowercase generally makes them more portable from one operating system to another.

Getting a Function's Name and Path

To obtain the file name for the function currently being executed, use the following function in your code.

```
mfilename
```

To include the path along with the file name, use

```
x``mfilename('fullpath')
```

For more information: See the mfilename function reference page.

What Files Does a Function Use?

For a simple display of all functions referenced by a particular function, follow the steps below:

1 Type clear functions to clear all functions from memory (see Note below).

2 Execute the function you want to check. Note that the function arguments you choose to use in this step are important, since you can get different results when calling the same function with different arguments.

3 Type inmem to display all MATLAB function files that were used when the function ran. If you want to see what MEX-files were used as well, specify an additional output, as shown here:

```
[mfiles, mexfiles] = inmem
```

Note clear functions does not clear functions locked by mlock. If you have locked functions, (which you can check using inmem), unlock them with munlock, and then repeat step 1.

Dependent Functions, Built-Ins, Classes

For a much more detailed display of dependent function information, use the `depfun` function. In addition to MATLAB function files, `depfun` shows which built-ins and classes a particular function depends on.

Function Arguments

Getting the Input and Output Arguments

Use `nargin` and `nargout` to determine the number of input and output arguments in a particular function call. Use `nargchk` and `nargoutchk` to verify that your function is called with the required number of input and output arguments.

```
function [x, y] = myplot(a, b, c, d)
disp(nargchk(2, 4, nargin))          % Allow 2 to 4 inputs
disp(nargoutchk(0, 2, nargout))      % Allow 0 to 2 outputs

x = plot(a, b);
if nargin == 4
   y = myfun(c, d);
end
```

Variable Numbers of Arguments

You can call functions with fewer input and output arguments than you have specified in the function definition, but not more. If you want to call a function with a variable number of arguments, use the `varargin` and `varargout` function parameters in the function definition.

This function returns the size vector and, optionally, individual dimensions:

```
function [s, varargout] = mysize(x)
nout = max(nargout, 1) - 1;
s = size(x);
for k = 1:nout
```

```
        varargout(k) = {s(k)};
    end
```

Try calling it with

```
[s, rows, cols] = mysize(rand(4, 5))
```

String or Numeric Arguments

If you are passing only string arguments into a function, you can use MATLAB command syntax. All arguments entered in command syntax are interpreted as strings.

```
strcmp string1 string1
ans =
     1
```

When passing numeric arguments, it is best to use function syntax unless you want the number passed as a string. The right-hand example below passes the number 75 as the string, '75'.

```
isnumeric(75)                    isnumeric 75
ans =                            ans =
   1                                0
```

For more information: See Command vs. Function Syntax in the MATLAB Programming Fundamentals documentation.

Passing Arguments in a Structure

Instead of requiring an additional argument for every value you want to pass in a function call, you can package them in a MATLAB structure and pass the structure. Make each input you want to pass a separate field in the structure argument, using descriptive names for the fields.

Structures allow you to change the number, contents, or order of the arguments without having to modify the function. They can also be useful when you have a number of functions that need similar information.

Passing Arguments in a Cell Array

You can also group arguments into cell arrays. The disadvantage over structures is that you do not have field names to describe each variable. The advantage is that cell arrays are referenced by index, allowing you to loop through a cell array and access each argument passed in or out of the function.

Program Development

In this section...

Planning the Program

When planning how to write a program, take the problem you are trying to solve and break it down into a series of smaller, independent tasks. Implement each task as a separate function. Try to keep functions fairly short, each having a single purpose.

Using Pseudo-Code

You may find it helpful to write the initial draft of your program in a structured format using your own natural language. This *pseudo-code* is often easier to think through, review, and modify than using a formal programming language, yet it is easily translated into a programming language in the next stage of development.

Selecting the Right Data Structures

Look at what classes and data structures are available to you in MATLAB and determine which of those best fit your needs in storing and passing your data.

For more information: see in the Programming Fundamentals documentation.

General Coding Practices

A few suggested programming practices:

- Use descriptive function and variable names to make your code easier to understand.

- Order subfunctions alphabetically in a file to make them easier to find.

- Precede each subfunction with a block of help text describing what that subfunction does. This not only explains the subfunctions, but also helps to visually separate them.

- Do not extend lines of code beyond the 80th column. Otherwise, it will be hard to read when you print it out.

- Use full Handle Graphics property and value names. Abbreviated names are often allowed, but can make your code unreadable. They also could be incompatible in future releases of MATLAB.

Naming a Function Uniquely

To avoid choosing a name for a new function that might conflict with a name already in use, check for any occurrences of the name using this command:

```
which -all functionname
```

For more information: See the which function reference page.

The Importance of Comments

Be sure to document your programs well to make it easier for you or someone else to maintain them. Add comments generously, explaining each major section and any smaller segments of code that are not obvious. You can add a block of comments as shown here.

```
%---------------------------------------------
% This function computes the ... <and so on>
%---------------------------------------------
```

For more information: See Comments in the MATLAB Programming Fundamentals documentation.

Coding in Steps

Do not try to write the entire program all at once. Write a portion of it, and then test that piece out. When you have that part working the way you want, then write the next piece, and so on. It is much easier to find programming errors in a small piece of code than in a large program.

Making Modifications in Steps

When making modifications to a working program, do not make widespread changes all at one time. It is better to make a few small changes, test and debug, make a few more changes, and so on. Tracking down a difficult bug in the small section that you have changed is much easier than trying to find it in a huge block of new code.

Functions with One Calling Function

If you have a function that is called by only one other function, put it in the same file as the calling function, making it a subfunction.

For more information: See "String Comparisons" on page 1-56 in the MATLAB Programming Fundamentals documentation.

Testing the Final Program

One suggested practice for testing a new program is to step through the program in the MATLAB debugger while keeping a record of each line that gets executed on a printed copy of the program. Use different combinations of inputs until you have observed that every line of code is executed at least once.

Debugging

The MATLAB Debug Functions

For a brief description of the main debug functions in MATLAB, type

```
help debug
```

For more information: See Debugging Process and Features in the
MATLAB Desktop Tools and Development Environment documentation.

More Debug Functions

Other functions you may find useful in debugging are listed below.

Function	Description
echo	Display function or script code as it executes.
disp	Display specified values or messages.
sprintf, fprintf	Display formatted data of different types.

Function	Description
whos	List variables in the workspace.
size	Show array dimensions.
keyboard	Interrupt program execution and allow input from keyboard.
return	Resume execution following a keyboard interruption.
warning	Display specified warning message.
MException	Access information on the cause of an error.
lastwarn	Return warning message that was last issued.

The MATLAB Graphical Debugger

Learn to use the MATLAB graphical debugger. You can view the function and its calling functions as you debug, set and clear breakpoints, single-step through the program, step into or over called functions, control visibility into all workspaces, and find and replace strings in your files.

Start out by opening the file you want to debug using **File > Open** or the open function. Use the debugging functions available on the toolbar and pull-down menus to set breakpoints, run or step through the program, and examine variables.

For more information: See Debugging Process and Features in the MATLAB Desktop Tools and Development Environment documentation.

A Quick Way to Examine Variables

To see the value of a variable from the Editor/Debugger window, hold the mouse cursor over the variable name for a second or two. You will see the value of the selected variable displayed.

Setting Breakpoints from the Command Line

You can set breakpoints with dbstop in any of the following ways:

- Break at a specific file line number.

- Break at the beginning of a specific subfunction.

- Break at the first executable line in a file.

- Break when a warning, or error, is generated.

- Break if any infinite or NaN values are encountered.

For more information: See Setting Breakpoints in the MATLAB Desktop Tools and Development Environment documentation.

Finding Line Numbers to Set Breakpoints

When debugging from the command line, a quick way to find line numbers for setting breakpoints is to use dbtype. The dbtype function displays all or part of the file, also numbering each line. To display delaunay.m, use

```
dbtype delaunay
```

To display only lines 35 through 41, use

```
dbtype delaunay 35:41
```

Stopping Execution on an Error or Warning

Use dbstop if error to stop program execution on any error and enter debug mode. Use dbstop if warning to stop execution on any warning and enter debug mode.

For more information: See "Debugging Process and Features" in the MATLAB Desktop Tools and Development Environment documentation.

Locating an Error from the Error Message

Click on the underlined text in an error message, and MATLAB opens the file being executed in its editor and places the cursor at the point of error.

For more information: See Finding Errors, Debugging, and Correcting MATLAB Files in the MATLAB Desktop Tools and Development Environment documentation.

Using Warnings to Help Debug

You can detect erroneous or unexpected behavior in your programs by inserting warning messages that MATLAB will display under the conditions you specify. See the section on Warning Control in the MATLAB Programming Fundamentals documentation to find out how to selectively enable warnings.

For more information: See the `warning` function reference page.

Making Code Execution Visible

An easy way to see the end result of a particular line of code is to edit the program and temporarily remove the terminating semicolon from that line. Then, run your program and the evaluation of that statement is displayed on the screen.

For more information: See Finding Errors, Debugging, and Correcting MATLAB Files in the MATLAB Desktop Tools and Development Environment documentation.

Debugging Scripts

Scripts store their variables in a workspace that is shared with the caller of the script. So, when you debug a script from the command line, the script uses variables from the base workspace. To avoid errors caused by workspace sharing, type `clear all` before starting to debug your script to clear the base workspace.

Variables

Rules for Variable Names

Although variable names can be of any length, MATLAB uses only the first N characters of the name, (where N is the number returned by the function `namelengthmax`), and ignores the rest. Hence, it is important to make each variable name unique in the first N characters to enable MATLAB to distinguish variables. Also note that variable names are case sensitive.

```
N = namelengthmax
N =
    63
```

For more information: See Naming Variables in the MATLAB Programming Fundamentals documentation.

Making Sure Variable Names Are Valid

Before using a new variable name, you can check to see if it is valid with the `isvarname` function. Note that `isvarname` does not consider names longer than `namelengthmax` characters to be valid.

For example, the following name cannot be used for a variable since it begins with a number.

```
isvarname 8thColumn
ans =
     0
```

For more information: See Naming Variables in the MATLAB Programming Fundamentals documentation.

Do Not Use Function Names for Variables

When naming a variable, make sure you are not using a name that is already used as a function name. If you do define a variable with a function name, you will not be able to call that function until you `clear` the variable from memory. (If it is a MATLAB built-in function, then you will still be able to call that function but you must do so using `builtin`.)

To test whether a proposed variable name is already used as a function name, use

```
which -all name
```

For more information: See Potential Conflict with Function Names in the MATLAB Programming Fundamentals documentation.

Checking for Reserved Keywords

MATLAB reserves certain keywords for its own use and does not allow you to override them. Attempts to use these words may result in any one of a number of error messages, some of which are shown here:

```
Error: Expected a variable, function, or constant, found "=".
Error: "End of Input" expected, "case" found.
Error: Missing operator, comma, or semicolon.
Error: "identifier" expected, "=" found.
```

Use the `iskeyword` function with no input arguments to list all reserved words.

Avoid Using i and j for Variables

MATLAB uses the characters i and j to represent imaginary units. Avoid using i and j for variable names if you intend to use them in complex arithmetic.

If you want to create a complex number without using i and j, you can use the complex function.

Avoid Overwriting Variables in Scripts

MATLAB scripts store their variables in a workspace that is shared with the caller of the script. When called from the command line, they share the base workspace. When called from a function, they share that function's workspace. If you run a script that alters a variable that already exists in the caller's workspace, that variable is overwritten by the script.

For more information: See Scripts in the MATLAB Programming Fundamentals documentation.

Persistent Variables

To get the equivalent of a static variable in MATLAB, use persistent. When you declare a variable to be persistent within a function, its value is retained in memory between calls to that function. Unlike global variables, persistent variables are known only to the function in which they are declared.

For more information: See Persistent Variables in the MATLAB Programming Fundamentals documentation.

Protecting Persistent Variables

You can inadvertently clear persistent variables from memory by either modifying the function in which the variables are defined, or by clearing the function with one of the following commands:

```
clear all
clear functions
```

Locking the file in memory with `mlock` prevents any `persistent` variables defined in the file from being reinitialized.

Global Variables

Use global variables sparingly. The global workspace is shared by all of your functions and also by your interactive MATLAB session. The more global variables you use, the greater the chances of unintentionally reusing a variable name, thus leaving yourself open to having those variables change in value unexpectedly. This can be a difficult bug to track down.

For more information: See Global Variables in the MATLAB Programming Fundamentals documentation.

Strings

In this section...
"Creating Strings with Concatenation" on page 10-29
"Comparing Methods of Concatenation" on page 10-29
"Store Arrays of Strings in a Cell Array" on page 10-30
"Converting Between Strings and Cell Arrays" on page 10-30
"Search and Replace Using Regular Expressions" on page 10-30

Creating Strings with Concatenation

Strings are often created by concatenating smaller elements together (e.g., strings, values, etc.). Two common methods of concatenating are to use the MATLAB concatenation operator ([]) or the sprintf function. The second and third line below illustrate both of these methods. Both lines give the same result:

```
numChars = 28;
s = ['There are ' int2str(numChars) ' characters here']
s = sprintf('There are %d characters here', numChars)
```

For more information: See "Creating Character Arrays" on page 1-35 and Converting from Numeric to String in the MATLAB Programming Fundamentals documentation.

Comparing Methods of Concatenation

When building strings with concatenation, sprintf is often preferable to [] because

- It is easier to read, especially when forming complicated expressions

- It gives you more control over the output format

- It often executes more quickly

You can also concatenate using the strcat function, However, for simple concatenations, sprintf and [] are faster.

Store Arrays of Strings in a Cell Array

It is usually best to store an array of strings in a cell array instead of a character array, especially if the strings are of different lengths. Strings in a character array must be of equal length, which often requires padding the strings with blanks. This is not necessary when using a cell array of strings that has no such requirement.

The `cellRecord` below does not require padding the strings with spaces:

```
cellRecord = {'Allison Jones'; 'Development'; 'Phoenix'};
```

For more information: See Cell Arrays of Strings in the MATLAB Programming Fundamentals documentation.

Converting Between Strings and Cell Arrays

You can convert between standard character arrays and cell arrays of strings using the `cellstr` and `char` functions:

```
charRecord = ['Allison Jones'; 'Development  '; ...
              'Phoenix      '];
cellRecord = cellstr(charRecord);
```

Also, a number of the MATLAB string operations can be used with either character arrays, or cell arrays, or both:

```
cellRecord2 = {'Brian Lewis'; 'Development'; 'Albuquerque'};
strcmp(charRecord, cellRecord2)
ans =
     0
     1
     0
```

For more information: See Converting to a Cell Array of Strings and String Comparisons in the MATLAB Programming Fundamentals documentation.

Search and Replace Using Regular Expressions

Using regular expressions in MATLAB offers a very versatile way of searching for and replacing characters or phrases within a string. See the help on these functions for more information.

Function	Description
regexp	Match regular expression.
regexpi	Match regular expression, ignoring case.
regexprep	Replace string using regular expression.

For more information: See "Regular Expressions" on page 2-71 in the MATLAB Programming Fundamentals documentation.

Evaluating Expressions

In this section...

Find Alternatives to Using eval

While the `eval` function can provide a convenient solution to certain programming challenges, it is best to limit its use. The main reason is that code that uses `eval` is often difficult to read and hard to debug. A second reason is that `eval` statements cannot always be translated into C or C++ code by the MATLAB Compiler.

If you are evaluating a function, it is more efficient to use `feval` than `eval`. The `feval` function is made specifically for this purpose and is optimized to provide better performance.

For more information: See MATLAB Technical Note 1103, "What Is the EVAL Function, When Should I Use It, and How Can I Avoid It?" at URL `http://www.mathworks.com/support/tech-notes/1100/1103.html`.

Assigning to a Series of Variables

One common pattern for creating variables is to use a variable name suffixed with a number (e.g., `phase1`, `phase2`, `phase3`, etc.). We recommend using a cell array to build this type of variable name series, as it makes code more readable and executes more quickly than some other methods. For example:

```
for k = 1:800
   phase{k} = expression;
end
```

Short-Circuit Logical Operators

MATLAB has logical AND and OR operators (&& and ||) that enable you to partially evaluate, or *short-circuit*, logical expressions. Short-circuit operators are useful when you want to evaluate a statement only when certain conditions are satisfied.

In this example, MATLAB does not execute the function myfun unless the file that defines myfun exists on the current path.

```
comp = (exist('myfun.m') == 2) && (myfun(x) >= y)
```

For more information: See "Short-Circuit Operators" on page 2-31 in the MATLAB Programming Fundamentals documentation.

Changing the Counter Variable within a for Loop

You cannot change the value of the loop counter variable (e.g., the variable k in the example below) in the body of a for loop. For example, this loop executes just 10 times, even though k is set back to 1 on each iteration.

```
for k = 1:10
   fprintf('Pass %d\n', k)
   k = 1;
end
```

Although MATLAB does allow you to use a variable of the same name as the loop counter within a loop, this is not a recommended practice.

MATLAB Path

In this section...
"Precedence Rules" on page 10-34
"File Precedence" on page 10-35
"Adding a Folder to the Search Path" on page 10-35
"Handles to Functions Not on the Path" on page 10-36
"Making Toolbox File Changes Visible to MATLAB" on page 10-36
"Making Nontoolbox File Changes Visible to MATLAB" on page 10-37
"Change Notification on Windows" on page 10-37

Precedence Rules

When MATLAB is given a name to interpret, it determines its usage by checking the name against each of the entities listed below, and in the order shown:

1 Variable

2 Nested Function

3 Subfunction

4 Private function

5 Class constructor

6 Overloaded method

7 MATLAB function file in the Current Folder

8 MATLAB function file on the path, or MATLAB built-in function

If you have two or more functions on the path that have the same name, MATLAB selects the function closest to the beginning of the path string.

For more information: See Function Precedence Order in the MATLAB Programming Fundamentals documentation.

File Precedence

If you refer to a file by its filename only (leaving out the file extension), and there is more than one file of this name in the folder, MATLAB selects the file to use according to the following precedence:

1 MEX-file

2 MDL-file (Simulink model)

3 P-Code file

4 MATLAB function (`.m`) -file

For more information: See Multiple Implementation Types in the MATLAB Programming Fundamentals documentation.

Adding a Folder to the Search Path

To add a folder to the search path, use either of the following:

- At the toolbar, select **File > Set Path**.
- At the command line, use the `addpath` function.

You can also add a folder and all of its subfolders in one operation by either of these means. To do this from the command line, use `genpath` together with `addpath`. The online help for the `genpath` function shows how to do this.

This example adds `/control` and all of its subfolders to the MATLAB path:

```
addpath(genpath('K:/toolbox/control'))
```

For more information: See Search Path in the MATLAB Desktop Tools and Development Environment documentation.

Handles to Functions Not on the Path

You cannot create function handles to functions that are not on the MATLAB path. But you can achieve essentially the same thing by creating the handles through a script file placed in the same off-path folder as the functions. If you then run the script, using run *path*/*script*, you will have created the handles that you need.

For example,

1 Create a script in this off-path folder that constructs function handles and assigns them to variables. That script might look something like this:

```
File E:/testdir/createFhandles.m
    fhset = @setItems
    fhsort = @sortItems
    fhdel = @deleteItem
```

2 Run the script from your Current Folder to create the function handles:

```
run E:/testdir/createFhandles
```

3 You can now execute one of the functions by means of its handle.

```
fhset(item, value)
```

Making Toolbox File Changes Visible to MATLAB

Unlike functions in user-supplied folders, MATLAB function files (and MEX-files) in the *matlabroot*/toolbox folders are not time-stamp checked, so MATLAB does not automatically see changes to them. If you modify one of these files, and then rerun it, you may find that the behavior does not reflect the changes that you made. This is most likely because MATLAB is still using the previously loaded version of the file.

To force MATLAB to reload a function from disk, you need to explicitly clear the function from memory using clear *functionname*. Note that there are rare cases where clear will not have the desired effect, (for example, if the file is locked, or if it is a class constructor and objects of the given class exist in memory).

Similarly, MATLAB does not automatically detect the presence of new files in *matlabroot*/toolbox folders. If you add (or remove) files from these folders, use `rehash toolbox` to force MATLAB to see your changes. Note that if you use the MATLAB Editor to create files, these steps are unnecessary, as the Editor automatically informs MATLAB of such changes.

Making Nontoolbox File Changes Visible to MATLAB

For functions outside of the toolbox folders, MATLAB sees the changes made to these files by comparing timestamps and reloads any file that has changed the next time you execute the corresponding function.

If MATLAB does not see the changes you make to one of these files, try clearing the old copy of the function from memory using `clear` *functionname*. You can verify that MATLAB has cleared the function using `inmem` to list all functions currently loaded into memory.

Change Notification on Windows

If MATLAB, running on Windows, is unable to see new files or changes you have made to an existing file, the problem may be related to operating system change notification handles.

Type the following for more information:

```
help changeNotification
help changeNotificationAdvanced
```

Program Control

In this section...

Using break, continue, and return

It is easy to confuse the break, continue, and return functions as they are similar in some ways. Make sure you use these functions appropriately.

Function	**Where to Use It**	**Description**
break	for or while loops	Exits the loop in which it appears. In nested loops, control passes to the next outer loop.
continue	for or while loops	Skips any remaining statements in the current loop. Control passes to next iteration of the same loop.
return	Anywhere	Immediately exits the function in which it appears. Control passes to the caller of the function.

Using switch Versus if

It is possible, but usually not advantageous, to implement switch-case statements using if-elseif instead. See pros and cons in the table.

switch-case Statements	if-elseif Statements
Easier to read.	Can be difficult to read.
Can compare strings of different lengths.	You need strcmp to compare strings of different lengths.
Test for equality only.	Test for equality or inequality.

MATLAB case Evaluates Strings

A useful difference between switch-case statements in MATLAB and C is that you can specify string values in MATLAB case statements, which you cannot do in C.

```
switch(method)
    case 'linear'
        disp('Method is linear')
    case 'cubic'
        disp('Method is cubic')
end
```

Multiple Conditions in a case Statement

You can test against more than one condition with switch. The first case below tests for either a linear or bilinear method by using a cell array in the case statement.

```
switch(method)
    case {'linear', 'bilinear'}
        disp('Method is linear or bilinear')
    case (<and so on>)
end
```

Implicit Break in switch-case

In C, if you do not end each case with a break statement, code execution falls through to the following case. In MATLAB, case statements do not fall

through; only one `case` may execute. Using `break` within a `case` statement is not only unnecessary, it is also invalid and generates a warning.

In this example, if `result` is `52`, only the first `disp` statement executes, even though the second is also a valid match:

```
switch(result)
   case 52
      disp('result is 52')
   case {52, 78}
      disp('result is 52 or 78')
end
```

Variable Scope in a switch

Since MATLAB executes only one `case` of any `switch` statement, variables defined within one `case` are not known in the other `cases` of that `switch` statement. The same holds true for `if-elseif` statements.

In these examples, you get an error when `choice` equals `2`, because `x` is undefined.

```
   -- SWITCH-CASE --                    -- IF-ELSEIF --
switch choice
   case 1                        if choice == 1
      x = -pi:0.01:pi;              x = -pi:0.01:pi;
   case 2                        elseif choice == 2
      plot(x, sin(x));              plot(x, sin(x));
end                              end
```

Catching Errors with try-catch

When you have statements in your code that could possibly generate unwanted results, put those statements into a `try-catch` block that will catch any errors and handle them appropriately.

The example below shows a `try-catch` block within a function that multiplies two matrices. If a statement in the `try` segment of the block fails, control passes to the `catch` segment. In this case, the `catch` statements check the error message that was issued (returned in `MException` object, `err`) and respond appropriately:

```
try
   X = A * B
catch err
   errmsg = err.message;
   if(strfind(errmsg, 'Inner matrix dimensions'))
      disp('** Wrong dimensions for matrix multiply')
end
```

For more information: See "The try-catch Statement" on page 6-18 in the MATLAB Programming Fundamentals documentation.

Nested try-catch Blocks

You can also nest try-catch blocks, as shown here. You can use this to attempt to recover from an error caught in the first try section:

```
try
   statement1                 % Try to execute statement1
catch
   try
      statement2              % Attempt to recover from error
   catch
      disp 'Operation failed' % Handle the error
   end
end
```

Forcing an Early Return from a Function

To force an early return from a function, place a return statement in the function at the point where you want to exit. For example,

```
if <done>
   return
end
```

Save and Load

In this section...

Saving Data from the Workspace

To save data from your workspace, you can do any of the following:

- Copy from the MATLAB Command Window and paste into a text file.

- Record part of your session in a `diary` file, and then edit the file in a text editor.

- Save to a binary or ASCII file using the `save` function.

- Save spreadsheet, scientific, image, or audio data with appropriate function.

- Save to a file using low-level file I/O functions (`fwrite`, `fprintf`, ...).

For more information: See Saving the Current Workspace in the MATLAB Desktop Tools and Development Environment documentation, and "Writing to Text Data Files with Low-Level I/O" in the MATLAB Data Import and Export documentation.

Loading Data into the Workspace

Similarly, to load new or saved data into the workspace, you can do any of the following:

- Enter or paste data at the command line.

- Create a script file to initialize large matrices or data structures.

- Read a binary or ASCII file using `load`.

- Load spreadsheet, scientific, image, or audio data with appropriate function.

- Load from a file using low-level file I/O functions (`fread`, `fscanf`, ...).

For more information: See Loading a Saved Workspace and Importing Data in the Desktop Tools and Development Environment documentation, and "Importing Data" in the MATLAB Programming Fundamentals documentation.

Viewing Variables in a MAT-File

To see what variables are saved in a MAT-file, use `who` or `whos` as shown here (the `.mat` extension is not required). `who` returns a cell array and `whos` returns a structure array.

```
mydataVariables = who('-file', 'mydata.mat');
```

Appending to a MAT-File

To save additional variables to an existing MAT-file, use

```
save matfilename -append
```

Any variables you save that do not yet exist in the MAT-file are added to the file. Any variables you save that already exist in the MAT-file overwrite the old values.

Note Saving with the `-append` switch does not append additional elements to an array that is already saved in a MAT-file. See the example below.

In this example, the second `save` operation does not concatenate new elements to vector A, (making A equal to [1 2 3 4 5 6 7 8]) in the MAT-file. Instead, it replaces the 5 element vector, A, with a 3 element vector, also retaining all other variables that were stored on the first `save` operation.

```
A = [1 2 3 4 5];   B = 12.5;   C = rand(4);
save savefile;
```

```
A = [6 7 8];
save savefile A -append;
```

Save and Load on Startup or Quit

You can automatically save your variables at the end of each MATLAB session by creating a finish.m file to save the contents of your base workspace every time you quit MATLAB. Load these variables back into your workspace at the beginning of each session by creating a startup.m file that uses the load function to load variables from your MAT-file.

For more information: See the startup and finish function reference pages.

Saving to an ASCII File

When you save matrix data to an ASCII file using save -ascii, MATLAB combines the individual matrices into one collection of numbers. Variable names are not saved. If this is not acceptable for your application, use fprintf to store your data instead.

For more information: See "Writing to Delimited Data Files".

Files and Filenames

In this section...

Naming Functions

Files that define MATLAB functions must start with an alphabetic character, may contain any alphanumeric characters or underscores, and must be no longer than the maximum allowed file name length (returned by the function `namelengthmax`).

```
N = namelengthmax
N =
    63
```

Since variables must obey similar rules, you can use the `isvarname` function to check whether a filename (minus its `.m` file extension) is valid for a MATLAB function file.

```
isvarname mfilename
```

Naming Other Files

The names of other files that MATLAB interacts with (e.g., MAT, MEX, and MDL-files) follow the same rules as the MATLAB function files, but may be of any length.

Depending on your operating system, you may be able to include certain nonalphanumeric characters in your filenames. Check your operating system manual for information on valid filename restrictions.

Passing Filenames as Arguments

In MATLAB commands, you can specify a filename argument using the MATLAB command or function syntax. For example, either of the following are acceptable. (The .mat file extension is optional for save and load).

```
load mydata.mat          % Command syntax
load('mydata.mat')       % Function syntax
```

If you assign the output to a variable, you must use the function syntax.

```
savedData = load('mydata.mat')
```

Passing Filenames to ASCII Files

ASCII files are specified as follows. Here, the file extension is required.

```
load mydata.dat -ascii            % Command syntax
load('mydata.dat','-ascii')       % Function syntax
```

Determining Filenames at Run-Time

There are several ways that your function code can work on specific files without you having to hardcode their filenames into the program. You can

- Pass the filename in as an argument

```
function myfun(datafile)
```

- Prompt for the filename using the input function

```
filename = input('Enter name of file:  ', 's');
```

- Browse for the file using the uigetfile function

```
[filename, pathname] =
   uigetfile('*.mat', 'Select MAT-file');
```

For more information: See the input and uigetfile function reference pages.

Returning the Size of a File

Two ways to have your program determine the size of a file are shown here.

```
      -- METHOD #1 --                        -- METHOD #2 --
  s = dir('myfile.dat');        fid = fopen('myfile.dat');
  filesize = s.bytes            fseek(fid, O, 'eof');
                                filesize = ftell(fid)
                                fclose(fid);
```

The dir function also returns the filename (s.name), last modification date
(s.date), and whether or not it is a folder (s.isdir).

(The second method requires read access to the file.)

For more information: See the fopen, fseek, ftell, and fclose function
reference pages.

Input/Output

In this section...

For more information and examples on importing and exporting data, see *MATLAB Data Import and Export*.

File I/O Function Overview

For a good overview of MATLAB file I/O functions, use the online "Functions — Categorical List" reference. In the Help browser **Contents**, select **MATLAB > Functions — Categorical List**, and then click **Data Import and Export**.

Common I/O Functions

The most commonly used, high-level, file I/O functions in MATLAB are save and load. For help on these, type doc save or doc load.

Functions for I/O to text files with delimited values are textscan, dlmread, dlmwrite.

For more information: See Text Files in the MATLAB "Functions — Categorical List" reference documentation.

Readable File Formats

Type doc fileformats to see a list of file formats that MATLAB can read, along with the associated MATLAB functions.

Using the Import Wizard

A quick method of importing text or binary data from a file (e.g., Excel® files) is to use the MATLAB Import Wizard. Open the Import Wizard with the command, uiimport filename or by selecting **File > Import Data** at the Command Window.

Specify or browse for the file containing the data you want to import and you will see a preview of what the file contains. Select the data you want and click **Finish**.

For more information: See "Tips for Using the Import Wizard" in the MATLAB Data Import and Export documentation.

Loading Mixed Format Data

To load data that is in mixed formats, use textscan instead of load. The textscan function lets you specify the format of each piece of data.

If the first line of file mydata.dat is

```
Sally    12.34 45
```

Read the first line of the file as a free format file using the % format:

```
fid = fopen('mydata.dat');
c = textscan(fid, '%s %f %d', 1);
fclose(fid);
```

returns

```
c =
    {1x1 cell}    [12.3400]    [45]
```

Reading Files with Different Formats

Attempting to read data from a file that was generated on a different platform may result in an error because the binary formats of the platforms may differ. Using the fopen function, you can specify a machine format when you open the file to avoid these errors.

Interactive Input into Your Program

Your program can accept interactive input from users during execution. Use the input function to prompt the user for input, and then read in a response. When executed, input causes the program to display your prompt, pause while a response is entered, and then resume when the **Enter** key is pressed.

Starting MATLAB

Getting MATLAB to Start Up Faster

Here are some things that you can do to make MATLAB start up faster.

- Make sure toolbox path caching is enabled.

- Make sure that the system on which MATLAB is running has enough RAM.

- Choose only the windows you need in the MATLAB desktop.

- Close the Help browser before exiting MATLAB. When you start your next session, MATLAB will not open the Help browser, and thus will start faster.

- If disconnected from the network, check the LM_LICENSE_FILE variable. See http://www.mathworks.com/support/solutions/data/1-17VEB.html for a more detailed explanation.

For more information: See Toolbox Path Caching in MATLAB in the MATLAB Desktop Tools and Development Environment documentation.

Operating System Compatibility

Executing O/S Commands from MATLAB

To execute a command from your operating system prompt without having to exit MATLAB, precede the command with the MATLAB ! operator.

On Windows, you can add an ampersand (&) to the end of the line to make the output appear in a separate window.

For more information: See Running External Programs in the MATLAB Desktop Tools and Development Environment documentation, and the system and dos function reference pages.

Searching Text with grep

grep is a powerful tool for performing text searches in files on UNIX systems. To grep from within MATLAB, precede the command with an exclamation point (!grep).

For example, to search for the word warning in all MATLAB function files of the Current Folder, ignoring case, you would use

```
!grep -i 'warning' *.m
```

Constructing Paths and Filenames

Use the fullfile function to construct path names and filenames rather than entering them as strings into your programs. In this way, you always get the correct path specification, regardless of which operating system you are using at the time.

Finding the MATLAB Root Folder

The `matlabroot` function returns the location of the MATLAB installation on your system. Use `matlabroot` to create a path to MATLAB and toolbox folders that does not depend on a specific platform or MATLAB version.

The following example uses `matlabroot` with `fullfile` to return a platform-independent path to the `general` toolbox folder:

```
fullfile(matlabroot,'toolbox','matlab','general')
```

Temporary Directories and Filenames

If you need to locate the folder on your system that has been designated to hold temporary files, use the `tempdir` function. `tempdir` returns a string that specifies the path to this folder.

To create a new file in this folder, use the `tempname` function. `tempname` returns a string that specifies the path to the temporary file folder, plus a unique filename.

For example, to store some data in a temporary file, you might issue the following command first.

```
fid = fopen(tempname, 'w');
```

Demos

Demos Available with MATLAB

MATLAB comes with a wide array of visual demonstrations to help you see the extent of what you can do with the product. To start running any of the demos, simply type `demo` at the MATLAB command prompt. Demos cover the following major areas:

- MATLAB
- Toolboxes
- Simulink
- Blocksets
- Real-Time Workshop®
- Stateflow®

For more information: See Demos in the Help Browser in the MATLAB Desktop Tools and Development Environment documentation, and the `demo` function reference page.

For More Information

In this section...
"Current CSSM" on page 10-55
"Archived CSSM" on page 10-55
"MATLAB Technical Support" on page 10-55
"Tech Notes" on page 10-55
"MATLAB Central" on page 10-55
"MATLAB Newsletters (Digest, News & Notes)" on page 10-55
"MATLAB Documentation" on page 10-56
"MATLAB Index of Examples" on page 10-56

Current CSSM

```
http://www.mathworks.com/matlabcentral/newsreader
```

Archived CSSM

```
http://mathforum.org/kb/forum.jspa?forumID=80
```

MATLAB Technical Support

```
http://www.mathworks.com/support/
```

Tech Notes

```
http://www.mathworks.com/support/tech-notes/list_all.html
```

MATLAB Central

```
http://www.mathworks.com/matlabcentral/
```

MATLAB Newsletters (Digest, News & Notes)

```
http://www.mathworks.com/company/newsletters/index.html
```

MATLAB Documentation

```
http://www.mathworks.com/access/helpdesk/help/helpdesk.html
```

MATLAB Index of Examples

```
http://www.mathworks.com/access/helpdesk/help/techdoc/demo_example.shtml
```

Index

Symbols and Numerics